# Harnessing Simulation Acceleration to Solve the Digital Design Verification Challenge

**by**

**Debapriya Chatterjee**

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

Doctoral Committee:

Associate Professor Valeria M. Bertacco, Chair
Professor Todd M. Austin
Professor Igor L. Markov
Assistant Professor Zhengya Zhang

To my parents

# Acknowledgments

I would like to thank my advisor Professor Valeria Bertacco, who introduced me to research. Her willingness to let me explore new ideas has been a cherished freedom, while our discussions on research directions have enabled me to pursue a fixed direction in an interesting and uncharted research territory. Moreover, my writing and presentation skills have grown significantly as a result of her mentoring and her careful attention.

I am also grateful for my committee members. I am appreciative of Professor Igor Markov's contributions: often pointing me to relevant research articles as well as our friendly interactions in the department and various conferences. Professor Todd Austin has been a consistent source of extremely valuable feedback on various matters throughout my studies. I am grateful to Professor Zhengya Zhang for his kind feedback.

Early in my graduate school career, I was very fortunate to work with Ilya Wagner and Andrew DeOrio – they instilled the fundamentals of the role of a graduate student into me. I am especially grateful to Joseph Greathouse, Andrea Pellegrini and Biruk Mammo for the innumerable illuminating discussions we have had in the office, often extending into late hours. I am indebted to Ronny Morad, Amir Nahir, Avi Ziv and Anatoly Koyfman from IBM Research, Haifa for allowing me to engage in a very close collaboration with the industry. I am really appreciative of the people I have collaborated with on many projects over the years: Andrew DeOrio, Calvin McCarter, Biruk Mammo, Chang-Hong Hsu, Doowon Lee, Sara Vinco, Nicola Bombieri, Ronny Morad, Amir Nahir, Avi Ziv, Anatoly Koyfman, Raviv Gal, Dmitry Pidan and Professor Franco Fummi. I am also appreciative of the interesting and insightful discussions with my colleagues Rawan Abdel-Khalek and Ritesh Parikh.

I would like to thank Dhanajay Anand, my friend and apartment mate for the entire duration of my stay in Ann Arbor, who supported me through thick and thin. I would also like to thank my friend from college days, Shaunak Chatterjee, for motivating me on many occasions. Finally, I would like to thank my parents: for providing me emotional support over half a decade halfway across the globe.

# Preface

Today, design verification is by far the most resource and time-consuming activity in the hardware development process for complex commercial integrated circuit designs such as microprocessor and system-on-chip (SoC) designs. Within this area, the vast majority of the verification effort in industry relies on simulation platforms, which can be implemented either in hardware or software. A "simulator" includes a model of each component of a design and has the capability of simulating its behavior under any input scenario provided by an engineer. Thus, simulators are deployed to evaluate the behavior of a design under as many input scenarios as possible and to identify and debug all incorrect functionality. Two features are critical in simulators for the validation effort to be effective: performance and checking/debugging capabilities. A wide range of simulator platforms are available today: on one end of the spectrum there are software-based simulators, providing a very rich software infrastructure for checking and debugging the design's functionality, but executing only at 1-10 simulation cycles per second (compare this to actual silicon chips, which execute at GHz speeds). At the other end of the spectrum, there are hardware-based platforms, such as accelerators, emulators and even prototype silicon chips, providing higher performances by 4 to 9 orders of magnitude, at the cost of very limited or non-existent checking/debugging capabilities. As a result, today, simulation-based validation is crippled: one can either have satisfactory performance on hardware-accelerated platforms or critical infrastructures for checking/debugging on software simulators, but not both.

This dissertation brings together the two ends of this spectrum by providing high performance solutions for software-based platforms and quality checking/debugging capabilities for hardware-based verification systems. To this end, the dissertation uses a two-pronged approach: it infuses performance into software simulators, and it brings in checking and debugging capabilities into hardware-based platforms. Specifically, it addresses the performance challenge of software simulators by leveraging inexpensive off-the-shelf graphics processors as a massively parallel execution substrate, and then exposing to it the parallelism inherent in the design model. The outcome is a simulation solution that achieves an order of magnitude speedup over traditional software-based simulation. For hardware-

based platforms, the dissertation provides solutions that offer enhanced checking and debugging capabilities by abstracting the relevant data to be logged during simulation so to minimize the cost of collection, transfer and processing. Observability for checking/debugging on these platforms is improved with a state restoration solution that is capable of reconstructing a number of signals while observing only a small fraction of them. Along with improved observability, the dissertation also brings in a number of high quality checking capabilities, comparable to those of their software-based simulation counterparts, while only imposing minimal logic or performance overhead ($\sim$20%). Altogether, the contributions of this dissertation enable effective high-performance simulation-based validation.

# Table of Contents

# List of Figures

**Figure**

# List of Tables

**Table**

# Chapter 1
# Introduction

Digital integrated circuits (IC) are pervasive in the modern world. Without digital IC's we would not have the broad range of today's consumer electronics: smartphones, tablets, personal computers (PC), teleconference systems, gaming consoles, interactive systems such as Microsoft Kinect [68] *etc.* Computers are built on several such digital chips; they include a microprocessor at their heart, and have become one of the most indispensable machines in modern human civilization. They form the backbone of the enterprise systems that run banking and stock markets for business and commerce; they collectively enable the Internet by operating in networked structures. All forms of transportation, such as automobiles, trains and airplanes are dependent on computer systems for propulsion, control and navigation.

Since digital systems have become an indispensable part of our lives, almost always deployed at the heart of activities that are critical for our safety and for the functioning of our society, it is critical that they are devoid of design flaws. However, these designs are conceived and developed by humans and design errors are unavoidable. Such design errors are known as functional bugs, as they deviate the function of a design from the ideal behavior. As several recent microprocessor manufacturer errata documents indicate [5, 49, 50, 48, 51], many functional bugs are often detected after the release of the product, and thus are present in almost every computer currently in use. The fallout of some functional bugs can be avoided via software workarounds. However, the impact of a critical functional bug released in the final product can be catastrophic. A malfunctioning system can cause financial loss, computer security breach or even loss of human life. A buggy product released in the field can cause irreparable damage to the reputation of a company and even jeopardize its survival due to the cost of product recalls. For example, the infamous Pentium FDIV bug was discovered in 1994. This bug caused some floating point division operations to compute a wrong result. Ultimately, the defective processors were recalled at a cost of $475 million for the manufacturer [65]. A similar issue today would cost at least 5 times as much due to faster ramp up timelines. Hence, it is imperative to

perform rigorous verification on a digital design to minimize the exposure to catastrophic situations.

The problem of verifying the functionality of digital designs during the development process has become increasingly challenging. Modern computer chips are vastly complex systems comprising billions of tiny transistors. Shrinking transistor sizes over each technology generation has lead to doubling of transistor count in a digital design every 18–24 months as predicted by Moore's law. With exponentially rising transistor count, digital designs can fit more logic. Designers take advantage of this trend by deploying more complex functionality into the design, as well as integrating a growing number of design components. This phenomenon has resulted in a proportional increase in verification and design debugging effort. For the past decade, the verification of digital designs has consumed about 70% of the time and effort dedicated to the development process. In 2008, a 16 core chip by Sun Microsystems required 100 person-years of verification [91]; some estimates by Intel corporation for their own development flow are in the thousands of person-years [87]; and efforts are rising over time. As we have now entered the era of mobile computing, new chips are released approximately every 6 months, exacerbating the verification problem.

Simulation-based validation is the primary workhorse in digital design houses. Simulation entails exercising an abstract model of a design with appropriate stimuli. A design is simulated at various levels of abstraction throughout the design process to validate different aspects of correctness. Appropriate validation coverage metrics are chosen to reflect what fraction of possible design behaviors have been exercised via simulation and subsequently validated. The more we explore the design behavior space, the higher the degree of coverage and our confidence in design correctness. Clearly, the rate of design behavior space exploration is proportional to simulation performance. Hence, to generate a desired level of validation coverage for a design in a shorter time frame it is imperative to increase the performance of simulation, and thus this aspect remains an area of active research. Naturally, verification engineers are resorting to high-performance simulation platforms; however, due to the very nature of such platforms, harnessing their simulation performance for efficient design checking and debugging is a problem which is unresolved today.

## 1.1 Importance of verification in the design flow

Digital integrated circuit designs are one of the most complex artifacts yet created by mankind. Typical designs, such as modern microprocessors, consist of billion of tran-

sistors (2.3 billion for the recent Nehalem-Ex [46] processor from Intel). Such complex designs would have been impossible to develop without a design flow that uses many layers of abstractions to harness the design process. First of all, specifications for a new design are obtained. Then designers develop the chip through several levels of abstraction: first

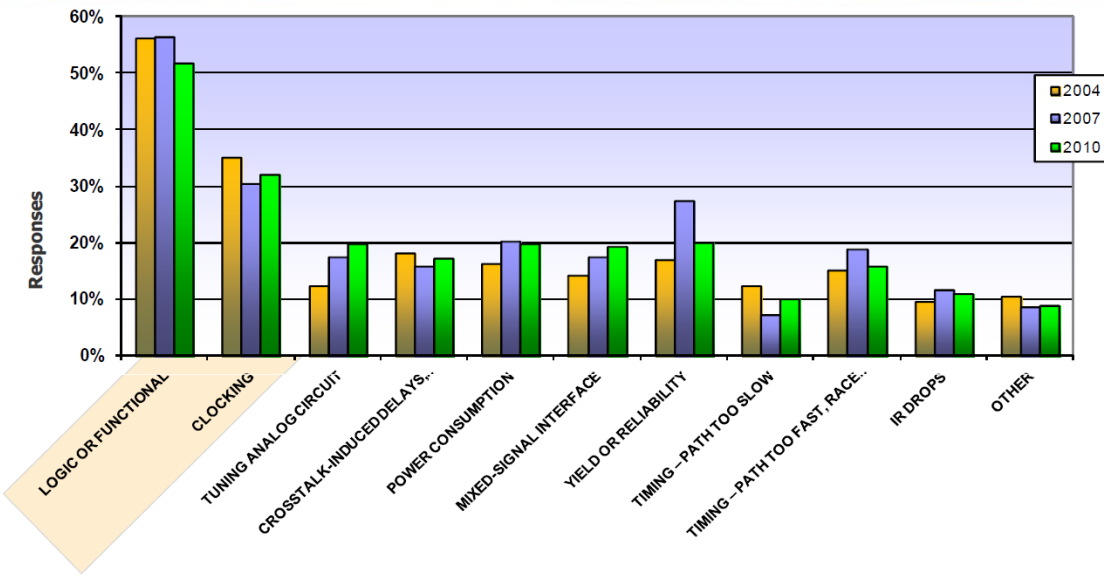



**Figure 1.1 Functional bugs are more prevalent than any other type of bug**, according to a verification study published by Wilson research group and Mentor Graphics [40]. This graph is based on the number of responses from verification engineers in the field on the question whether they have encountered a particular type of bug. (Reproduced with permission from Harry Foster of Mentor Graphics)

The process of verification is intricately intertwined with the design development process. The main purpose of verification is to ensure that the specifications are met at each layer of abstraction and for the final silicon chip itself. One of the most important classes of verification in this regard is functional verification. Functional bugs, which prevent the design from working as specified, usually occur due to human errors either in the functional modeling itself due to wrong interpretation of the specification, or while developing the register transfer level (RTL) model in a HDL, due to erroneous behavioral modeling. There can be inconsistencies in the specification itself, which will also result in erroneous behavior. Functional verification attempts to ensure that the functionality of a logic design is as it was intended in the specifications, and attempts to detect functional bugs. It may even detect inconsistencies in the specification after the design is available. This dissertation focuses on enhancing the performance of functional verification and the solutions presented are primarily concerned with detection and diagnosis of functional bugs. Indus-

try verification experiences [15, 53] indicate that upwards of 43% of the total bugs in a microprocessor design are functional bugs. A more recent verification study published by Wilson research group and Mentor Graphics [40], reveals that functional bugs are more prevalent in a typical design than any other type of bug (see Figure 1.1).

Even though functional bugs are typically introduced early in the design process, they may not be detected until silicon prototypes are tested, or even after product deployment. Most of these bugs are detected during extensive functional verification using simulation or other verification methods early in the RTL design phase. The simulation stimuli are provided by a testbench connected to the design model. At this stage, eliminating a bug involves first understanding the issue by studying simulation traces. Then, the error can be fixed by modifying the RTL source code. However, due to performance limitations of software-based simulation, only relatively simple and short tests can be performed at this stage. When designing large integrated digital systems, which are increasingly common due to proliferation of system-on-chips (SoC) deployed in smartphones and tablets, this can be a serious limitation, as system-level tests cannot be executed. The impact of a bug caught in this phase may be limited to a schedule delay of a few months.

Further into the design process, hardware-accelerated simulation platforms, such as accelerators and emulators come into play. These platforms offer orders of magnitude better simulation performance than software-based simulators; however, checking and debugging is not as straightforward as software-based simulation. The simulation performance of these platforms allow for executing much more complex and long tests on a design, which is critical to obtain more coverage on design correctness. If an erroneous design behavior is found at this stage, tracing it back to its root cause requires substantially more effort than the previous stage of software-based simulation, as checking and debugging capabilities are limited. On the positive side, a functional bug detected at this phase can still be remedied by modifying the RTL source code.

As the design process continues, the physical design steps of technology mapping, placement and layout are performed, and early silicon prototypes are manufactured for fast, at-speed testing. A functional bug that escapes pre-silicon functional verification can still be detected at this stage. Electrical and transistor faults can also manifest at this stage. However, failures identified at this stage require re-tooling the manufacturing process for a modified design, called a re-spin. Re-spins may require several months of delay and are very expensive due to the high re-tooling cost, which can range from approximately $3 million to $30 million.

Following silicon prototype testing and necessary re-spins, a new chip can be shipped to customers. At this late stage, failures have widespread and critical impact. A recall on a

| Bug | Year | Description | Aftermath |
|---|---|---|---|
| Intel Pentium FDIV bug | 1994 | This bug caused some floating point division operations to compute the wrong result, affecting approximately 1 in 9 billion possible divisions. | Ultimately, the defective processors were recalled at a cost of $475 million [65]. |
| Intel Pentium F00F bug | 1997 | As a result of this bug, execution of a certain instruction would put the processor in such a state that it stops servicing any interrupt and it must be reset to recover [1]. | A later stepping of the processor fixed the bug. |
| AMD Athlon X2 bootup bug | 2002 | A cache coherency bug between the 2 cores resulted in some speculative write operations to not to be seen by the other core. | Significant delay during the boot up process. |
| AMD Phenom TLB bug | 2007 | A bug in the TLB can lead to a race condition and subsequent system lockup. | A BIOS fix was issued; however, it limited the performance of this chip to the levels of its predecessor. |
| Intel 82574L Ethernet controller bug | 2013 | Sending a specially crafted packet to some Intel 82574L Ethernet controllers can cause the hardware to hang, and the "packet of death" could be put to malicious use and crash systems even when protected by a firewall [47]. | This bug can be fixed by reprogramming the EEPROM in the chip; however, many computer systems across the world still remain vulnerable. |

**Table 1.1 Real world impact of functional bugs**.

faulty product can take up to a year, at which point newer, competing products may already be available. The life of a company can be jeopardized by loss of reputation due to failures in the field, as well as associated costs of recall. A list of functional bugs released into the final product over last few decades and their aftermath is presented in Table 1.1.

A product released with such functional bugs can cause immense damage to the reputation of a company, and can also have severe impact on the customer. However, as design complexity increases with the advent of many-core microprocessor designs, the number of functional bugs released in final product show an upward trend. This is illustrated in Figure 1.2, which is compiled from data [49, 50] released by Intel corporation reporting information on the of bugs discovered in some Intel processor products after the product release date. Evidently, more recent products have a higher number of bugs escaped in the field, as well as a higher rate of additional bugs exposed after the initial release.

In view of the consequences of a functional bug escaped in the final product, functional verification has become a very critical component of the digital design flow. This is re-

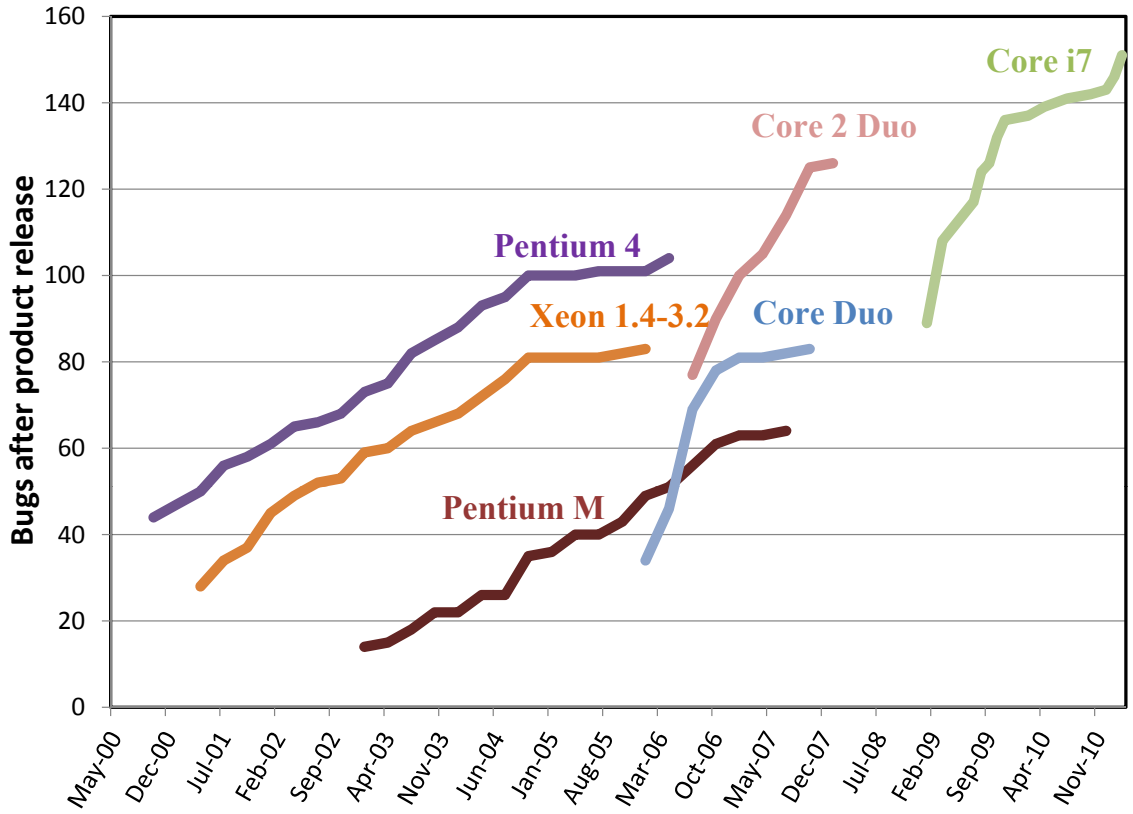



**Figure 1.2 Number of bugs discovered after product release for recent Intel microprocessors**.

flected in current industry trends; during the 2007-2010 period, the number of verification engineers in the semiconductor industry has increased by 58% compared to a mere 4% for design engineers [40]. In fact up to 70% of the effort in a modern microprocessor design project is dedicated to verification, of which functional validation claims the lion's share [15]. The earlier in the design flow a bug is detected and diagnosed, the less is the associated cost to correct it. As a result, improving the performance of verification without sacrificing quality is bound to have have a direct positive impact on shortening the length of the design cycle and on providing higher confidence in the correctness of a product.

## 1.2 Phases of functional verification

The process of verifying the correctness of a digital design continues along the entirety of the design process. Pre-silicon verification is performed on different abstractions of the design model to ensure that the design meets the specifications at all levels of abstraction. When silicon prototypes are available, post-silicon validation is performed to detect the design bugs that escaped pre-silicon verification, as well as to detect other design failures.

All in all, the goal of design verification is to ensure that the final product strictly obeys the design specifications.

### 1.2.1 Pre-silicon verification

Pre-silicon verification is applied on different abstractions of the design model. Typically most functional design bugs are discovered and corrected during this phase. The main advantage of diagnosing a functional design bug in pre-silicon phase is that engineers may only need to correct the RTL code describing a design, while in post-silicon phase it will need a re-spin of the whole chip: an exorbitantly costly process. The pre-silicon verification techniques fall under three broad categories: formal, simulation-based and semi-formal.

**Formal verification** techniques can mathematically prove that a certain property holds for a design. The main advantage of formal verification is that it is a complete method, e.g. if a property is proven by model checking then it holds for all possible execution scenarios of the design. Formal techniques implicitly enumerate all possible states of a design with the aid of binary decision diagrams [22] or convert property checking into Boolean satisfiability (SAT) instances and deploy a SAT solver [69]. As a result, for large designs with large number of sequential elements, the state explosion problem can happen; which leads to exponential memory requirements for the decision diagram representation, rendering the formal tool useless. The same problem can also manifest in the SAT solver as exponential runtime, when the SAT solving algorithm needs to actually explore exponentially large solution space before reaching a decision. Additionally, formal verification requires a large amount of human effort to precisely state the properties to be proven and writing a complete set of formal properties for a large design can require as much effort as the design process itself.

Currently formal verification is primarily applied either on individual components of a large design such as floating point units of a micro-processor [15] or on abstracted representations of a design, such as the Mur$\varphi$ [35] tool for verifying cache coherence protocols. Another approach to pre-silicon verification that is closely related with formal verification is known as assertion-based verification, where certain behavioral properties are attached to the relevant parts of the behavioral description of a design from the very beginning of design process. Modern hardware description languages support assertions as a part of the language framework, e.g., SystemVerilog provides SystemVerilog Assertions (SVA). These properties can either be decided formally, or a counter-example could be found with simulation or semi-formal methods.

**Simulation-based validation** is used as the primary workhorse in the industry to detect functional design bugs and for debugging the design under development. A model of the design (expressed in behavioral, RTL or structural logic gate-level abstraction) is simulated using either hand-written testbenches or constrained-random inputs. Typical hardware description languages (such as Verilog, VHDL) provide means of writing testbenches and several verification languages (*e*, Vera, SystemVerilog) provide mechanisms to provide constrained-random inputs to an interfaced design. A comprehensive overview of functional validation using simulation can be found in [94]. Simulation is an incomplete method, since all possible execution paths cannot be exhausted in the available development time window. However, validation engineers strive to simulate as many tests as possible within the available time frame to achieve higher coverage according to some pre-defined coverage metric (such as code / functional coverage). As a result, simulation performance is extremely important from a practical stand point.

The traditional simulation platform is "software-based simulation" where a simulation software executes on a general purpose processor in a workstation or server. Traditionally software-based simulation is primarily used for design time verification; however, on large industrial designs such simulation is extremely slow (1-10 clock cycles per second). As a result only a limited set of simple and short testcases are feasible to be validated by software-based simulation. From a design debug perspective software-based simulation is an excellent platform since any design signal value can be accessed by the validation engineer, which leads to a relatively easy debug process. Higher performance can be obtained from expensive hardware-accelerated platforms; however, it generally comes at a cost of signal observability and debugging ease. Presented with a limited design time window, it is imperative to use such platforms to simulate complex and long test regressions.

**Semi-formal verification** is essentially a hybrid of formal and simulation-based techniques. Simulation based state-space exploration with guidance from a formal engine is an example of such a technique. Verification tools such as Magellan [90] deploy such techniques to disprove a property by finding counter-examples through formal engine guided simulation.

### 1.2.2 Post-silicon validation

Post-silicon validation is only possible when the first silicon prototypes become available. The advantage in post-silicon phase is the fact that test execution speed can be same as

the chip itself, which is orders of magnitude faster than all forms of pre-silicon simulation. Hence, long programs such as operating systems or extensive constrained random tests can be executed. Test outcomes can then be validated with a mix of hardware assertions, comparison of test outputs against a golden model, or with the aid of self-checking mechanisms. These testcases can reach very deep states of a design or expose corner cases that are hard to reach with limited simulation performance of pre-silicon verification. Hence the hard to find functional bugs may be discovered during this phase of verification. However, on the downside, if a test failure indicates an error, it can be functional, electrical (process, logic or circuit related), or due to a manufacturing defect. Additionally, while pre-silicon modeling is deterministic, manufactured silicon circuits may have non-deterministic behavior. Verification engineers have access to very limited amount of debugging capability for tests on the silicon: such as on-chip logic analyzers, re-purposed design-for-test (DFT) features like scan chains and partially reconfigurable embedded checkers [3, 77]. As a result of very limited observability and controllability characteristics, post-silicon failure diagnosis and debug is an extremely challenging proposition.

## 1.3 Factors crippling simulation-based validation

In the industry, functional validation of digital designs has been traditionally performed with software-based simulation of the design description. A software application named design simulator, executes on a general purpose computer (such as a PC or a server), reads in files describing the design and the associated testbench, and then simulates the design as intended. Synopsys VCS or Cadence NC-verilog are examples of such simulation software. The golden output corresponding to correct design behavior is generated using a high-level model of the design or by other means and compared with the simulation output. A plethora of checking and debugging solutions exist for software-based simulation [94]. However, in the wake of ever larger and complex designs, and the long and intricate tests such designs necessitate, performance of software-based simulation is not even close to adequate for the verification need. Hence, verification engineers are increasingly adopting expensive hardware-accelerated simulation platforms for functional validation, and even performing functional validation in silicon prototypes of a design. Though these platforms offer high-performance simulation, checking and debugging capability is extremely limited.

### 1.3.1 Limited performance of software-based simulation

The available commercial software-based simulators can only deliver 1-10 simulation cycles per second for a full-chip simulation of typical current designs. At this speed, even a ten million cycle regression, fairly common in a typical micro-processor validation suit, will take an inordinate amount (more than a week) of time to simulate. A billion cycle regression will be completely infeasible. Clearly software-based simulation performance is far short of adequate. Hence, even though excellent checking and debugging capability exists for software-based simulation, they cannot be leveraged to perform high-quality validation. To exacerbate the problem, product cycles in the leading design houses are becoming shorter, making it infeasible to reach a desired level of verification coverage by software-based simulation alone. Expensive hardware-accelerated simulation platforms can provide higher performance. However, using these platforms can drive up the cost of the validation process. Under these circumstances, solutions for high-performance software-based simulation at a low cost are extremely desirable. It will contribute towards meeting product schedules while assuring design correctness by achieving functional coverage goals alongside adequate checking and debugging capability.

### 1.3.2 Limited validation capability for hardware-accelerated simulation

Hardware-accelerated platforms offer simulation speed in the range of few kHz to hundreds of MHz, though still short of silicon speed, they bring down the simulation time needed for even the longest of test regressions into the realm of feasible. However, as we move to hardware-accelerated platforms beyond traditional software-based simulation, the ease of checking and debugging is drastically diminished. An array of tools and checking/debugging methodologies has been built around software-based simulation over last few decades, where it is possible to have complete observability into the inner workings of a simulated design. While in hardware-accelerated simulation observability is at best partial and often comes at a cost of reduced simulation performance. The amount of information than can be gathered from these platforms per simulation cycle is limited as it comes with significant degradation of performance. Observability of internal signals is reduced to a great extent making debugging a challenge as well. Logic capacity related constraints on such platforms severely restrict the amount of additional logic dedicated for verification purposes. Even for acceleration platforms which does not have a strict logic capacity limit, increasing amount of simulated logic has an adverse effect on simulation performance. As

a result effective checking/debugging on these platforms poses a challenge.

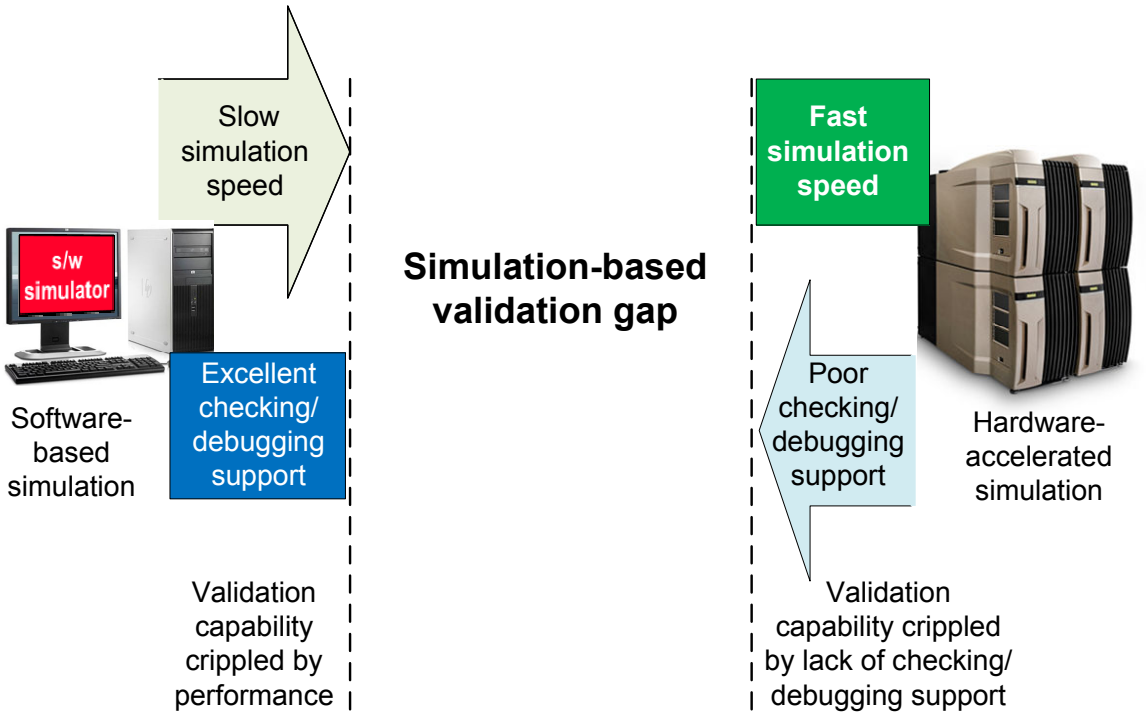



**Figure 1.3 The simulation-based validation gap.** Simulation performance and checking/debugging ease are not achieved together.

### 1.3.3 The simulation-based validation gap

The fundamental problem plaguing the current state-of-the-art simulation-based validation is depicted in Figure 1.3. On one hand we have traditional software-based simulation, which is equipped with high-quality checking and debugging capability. However poor simulation performance cripples its applicability to real world designs. On the other hand we have hardware-accelerated simulation platforms such as accelerators and emulators, where simulation performance is bountiful. However the same level of ease in checking and debugging is no longer available, and thus it fails to fully leverage the performance advantage to benefit the process of validation. This problem presents itself as a critical gap in the current state of simulation-based validation; we do not achieve high-quality checking and debugging capability along with high-performance simulation. The broad goal of my dissertation is to bridge this gap.

## 1.4 Overview of my dissertation

Digital designs face increasing complexity and tighter release schedules, challenging the ability of the current design process to deliver a correctly functioning product in a given time frame. Simulation-based validation is the primary method deployed in the industry to ensure design correctness. As explained in the previous section, the effectiveness of the current state of simulation-based validation is compromised by a critical gap; high-quality checking and debugging capability is not attained simultaneously with desirable simulation performance. This dissertation presents novel solutions to deliver low-cost high-performance software-based simulation as well as solutions to provide checking and debugging capability on hardware-accelerated simulation platforms, therefore bridging this gap from both ends. The solutions in this dissertation take on the complexity of modern designs by accelerating the simulation-based validation process as a whole. These solutions will enable verification practitioners to harness the potential of simulation to full extent; design checking and debugging will be achieved at much higher simulation performance than attainable currently. Ultimately this will enable achieving increased coverage for short product cycles, thereby decreasing the probability of occurrence of a bug in the final product, while conforming to tight release schedules.

### 1.4.1 Bridging the simulation-based validation gap

Figure 1.4 presents an overview of the solution proposed in this dissertation, which bridges the gap from both directions. One direction is to simply improve the performance of software-based simulation. This objective is achieved by altering the execution substrate of software-based simulation, whose performance is limited by serial execution in a general purpose processor, to a massively parallel platform such as general purpose graphics processing unit (GPGPU). Parallel execution can boost the performance of software-based simulation of digital designs to a great extent, since it possesses an inherently parallel computation pattern. Fortunately, these platforms are fairly inexpensive, thus allowing for a low-cost simulation acceleration solution. In this case, the simulator is still implemented as a software application, thus all checking and debugging solutions used in traditional software-based simulation can still be used with little or no modification.

The other direction is to craft validation schemes that attempt to leverage the performance of existing hardware-accelerated simulation platforms in an efficient fashion. These schemes attempt to provide high-quality checking and debugging capability under platform-specific constraints of signal observability and logic capacity. The problem of

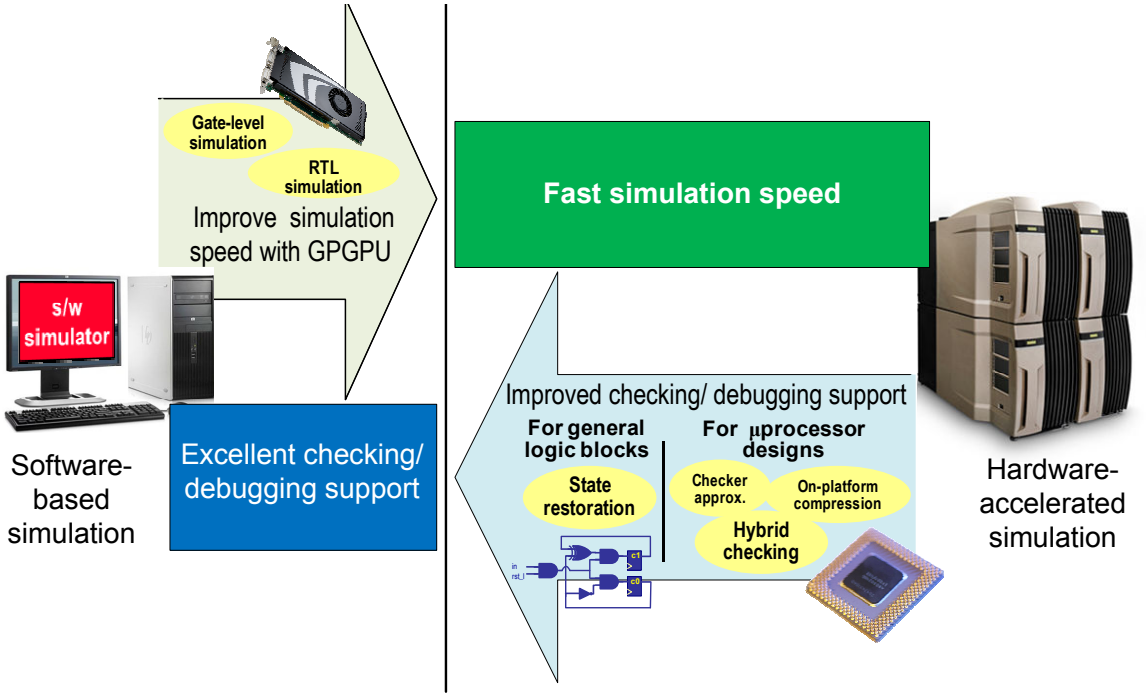



**Figure 1.4 Overview of the dissertation**. My solutions bridge the gap in the current state of simulation-based validation from both ends.

reduced signal observability in these platforms is tackled by reconstructing signal values from a limited number of observed signals; an approach applicable to general logic blocks, thus paving the way to better debugging. Checking capability is brought to these platforms by (i) delegating some of the design checking responsibility to lightweight embedded logic, (ii) compressing simulation data on-platform and checking the compact log post-simulation and (iii) a synergistic combination of both. The later set of schemes requires design knowledge and is well suited for validation of micro-processor designs.

### 1.4.2 Improving performance of software-based simulation

As mentioned earlier, one of the directions is toward delivering high-performance software-based logic simulation at a low cost. Logic simulation is used to validate designs at the behavioral level, as well as the structural level, ensuring that a synthesized circuit's netlist matches the functionality and timing of the behavioral model. Structural netlists are particularly cumbersome for simulation because of their low-level specification and the fine granularity of the structural definition, which consist of large number of gate primitives from the target technology library. Recent availability of general purpose computing programming models for high-performance and massively parallel GPUs led me to explore a

new simulation architecture targeting these hardware platforms, with the hope of delivering a conspicuous performance advantage at a small hardware cost (that of a GPU peripheral).

This effort has resulted in a simulation solution called GCS, which exposes the parallelism available in the simulation problem to the massively parallel processing hardware of the GPU, by using novel partitioning algorithms. GCS is able to deliver an order-of-magnitude performance improvement over traditional software-based simulators executing on general purpose processors. The research presented in this dissertation, further explores the applicability of such massively parallel processing into behavioral simulation as well. This reveals that an order of magnitude simulation performance improvement can be achieved for designs expressed in a behavioral subset of SystemC. These solutions provide a cost-effective means for simulation acceleration, allowing for higher validation coverage in an affordable fashion. Thus, they close in on the performance gap between software-based simulation and hardware-accelerated platforms.

### 1.4.3 Bringing in validation capability to hardware-accelerated platforms

Hardware-accelerated simulation platforms deployed in the industry offer 3-6 orders of magnitude simulation performance over software-based simulation, however as mentioned before these platforms do not provide the same degree of checking/debugging ease as a software-based simulator. This presents us with another gap in validation capability beyond traditional design-time software-based simulation. This dissertation bridges this gap by providing solutions that offer enhanced checking/debugging capability on these platforms while maintaining the performance advantage.

#### Observability for debugging beyond software-based simulation

One of the fundamental limitations of these platforms is that observability comes at the cost of performance loss or logic overhead, hence only a small subset of signals can be observed. In light of this problem, I focused towards achieving improved debugging capability in the wake of limited signal observability that is endemic to all hardware-accelerated platforms and post-silicon validation. Limited observability of internal signals of a design hinders the ability to diagnose and debug already detected bugs. A solution to address this issue leverages trace buffers: these are buffers embedded into the design with the goal of recording the value of a number of signals, over a time interval, triggered by a user-specified event. However, we can only record a small number of such signals due to the constraints of the

platform itself. A key observation in this regard is that the information content carried by those signals can be much larger as many other signal values can be reconstructed from the recorded information. This can be thought of as a lossless compression of a subset of signals into a very small number of recorded signals. Ideally, we would like to select signals enabling the maximum amount of reconstruction of internal signal values i.e. the most information content. To this end, an accurate restoration capacity metric is developed, and a novel algorithm striving to select a set of signals obtaining maximal reconstruction is delineated. This solution does not require any design specific knowledge as it operates on the structural description of any general logic block. It is able to provide a higher degree of reconstruction than previous solutions in the same space and thus paves the way towards better debugging capability beyond software-based simulation.

**Checking capability beyond software-based simulation**

The rest of my contributions are towards bringing in checking solutions that are currently only feasible with software-based simulation, to the realm of hardware-accelerated platforms; however, they require design specific knowledge and are mostly applicable to microprocessor designs.

The first of them is an attempt to bring in existing software-based checkers that are used with software-based simulation, into the purview of accelerated simulation. To this end, checkers must be transformed into synthesizable, compact logic blocks, yet with bug-detection capabilities similar to that of their software counterparts. The key idea in this research is named "*approximate checkers*", which trade off logic complexity with bug detection accuracy by leveraging novel techniques to approximate complex software checkers into small synthesizable hardware blocks which can be simulated along with a design on an hardware-accelerated platform. These approximate checkers are able to maintain a high degree of checking accuracy with small logic footprint.

In contrast to checker approximation, I also explored a *log-and then-check* approach to checking on hardware-accelerated platforms. This approach is useful for adapting those software-based checkers for hardware-accelerated platforms, which have a checking component that is complex enough that it cannot simply be converted into a hardware description. As discussed before, only very few signals can be recorded without degrading simulation performance, hence the checking methodology itself needs to be adapted to work with compressed and/or partial information. This concept is demonstrated with an important checking solution for microprocessor verification, namely instruction by instruction checking (IBI). This particular checking scheme tracks the architectural events

generated by a microprocessor design model when it is executing a test regression and compares this with a golden architectural model. The checking scheme is adapted to hardware-accelerated platforms via a novel solution where the data associated with events are compressed and logged using additional simulated hardware on-platform, and a software checker is created to operate on this compressed log of events post-simulation. This approach results in a solution that is almost as accurate as the entirely software-based solution, yet offers the same performance as the hardware-accelerated simulation platform provides.

Finally, my dissertation culminates in a unifying solution which brings together different ideas on performing checking beyond traditional software-based simulation for modern microprocessor designs. *Hybrid checking* attempts to combine the ideas of using lightweight embedded logic to perform checks during simulation as well as performing post-simulation checks on a compressed event log in a synergistic fashion. To this end, typical checks needed for a modern micro-processor design are separated into cycle-accurate local embedded assertions (implemented as lightweight embedded logic) and event-accurate functionality checks requiring a post-simulation checking phase. Embedded logic is further used to compress the data associated with events relevant to functionality checks.

Overall these solutions enable comprehensive simulation-based validation at the high-performance offered by hardware-accelerated simulation platforms, thus bridging the gap in validation capability beyond software-based simulation.

## 1.5 Organization of the dissertation

The remainder of the dissertation is organized as follows: Chapter 2 provides a more in-depth look into simulation-based validation, focusing on the different hardware accelerated platforms and platform-specific trade-offs. The low cost acceleration solutions for both structural and behavioral logic simulation are described in Chapter 3. These solutions leverage off-the-shelf GPUs to deliver orders-of-magnitude better performance than traditional software-based simulators executing on general purpose processors. These efforts bridge the performance gap between very expensive acceleration platforms and slow software-based simulation.

A signal selection algorithm for maximizing state restoration in general logic blocks, which attempts to provide increased observability from partial knowledge of signals for debugging purposes, is presented in Chapter 4. Approximate checkers enable bug-detection

while preserving acceleration performance; this solution is explained in Chapter 5. Compaction of trace data on-platform reduces the volume of data that has to be transferred off-platform; Chapter 5 also describes such a solution applied to instruction-by-instruction checking for micro-processor designs. Hybrid checking, which leverages both approaches presented in chapter 5, is described in Chapter 6. These solutions bridge the gap in validation capability beyond software-based simulation. Finally Chapter 7 summarizes the conclusions of this dissertation.

# Chapter 2
# The Simulation Spectrum

As discussed in the introduction, simulation is the primary mode of functional validation during the design process. All major design houses deploy large arrays of servers for software-based simulation runs of the design under development. Different types of hardware-accelerated simulation platforms such as acceleration platforms, emulators and prototyping platforms are becoming increasingly vital in coping with the vast effort demands of design validation.

A full spectrum of simulation-based validation platforms is available today: the differences among these platforms are primarily in simulation performance and ease of checking and debugging. When a design is in its initial developmental stages, software-based simulation is the only validation approach deployed. At later stages, acceleration platforms and emulators are heavily deployed for performing extensive regression testing. Late in the design process, FPGA-based prototyping platforms are also utilized to emulate the design at the full-system level. Finally, at-speed tests are possible on early silicon prototypes and this phase falls under the domain of post-silicon validation.

In this chapter, I overview the full spectrum of platforms and I identify and discuss a common set of challenges that must be addressed to attain effective simulation-based validation beyond traditional software-based simulation. Finally, I introduce the solutions proposed by this dissertation to overcome the challenges outlined. The chapter is organized as follows: first, the characteristics of a broad range of platforms are discussed, along with their associated trade-offs. Second, the existing approaches in the space of simulation-based validation are described, along with their shortcomings. From this discussion, a set of key challenges that need to be overcome to enable high-performance simulation-based validation are identified. Then, I demonstrate how the solutions presented in this dissertation promise to solve each of the key challenges and pave the way towards effective, high-performance simulation-based validation.

## 2.1 Spectrum of validation platforms

Through the various phases of the design process, the fundamental method of simulation-based validation remains unchanged, while the platforms on which it is applied may vary. The basic method consists of running a test on a model of the design, while the expected outputs are provided by a reference model (a.k.a., golden model). Any mismatch between the outputs of the design and those of the golden model is an indicator of a possible functional bug. The viable length and complexity of the test depends on the simulation performance of the associated platform. The verification engineer's responsibility is to debug the design's failing tests, (that is, those flagging a mismatch) and identify the root-cause the problem. Any solution that helps in pinpointing the problem is a debugging aid. The spectrum of simulation platforms and their associated characteristics are summarized in Figure 2.1 in order of increasing performance, along with their checking and debugging capabilities.

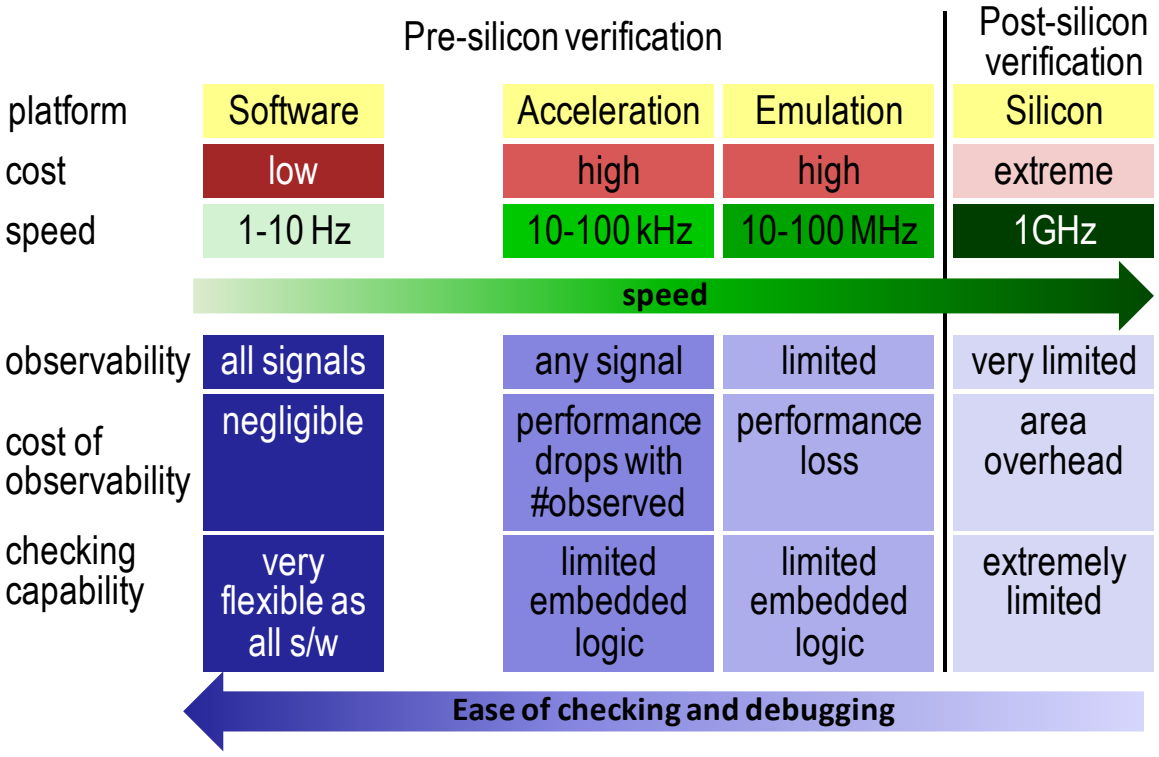



**Figure 2.1** **The simulation spectrum**. The key characteristics of each major simulation platforms are outlined, along with a qualitative evaluation of their checking and debugging capabilities.

### 2.1.1 Software-based simulation

Software-based simulation is by far the most dominant design-aid since it provides the most matured infrastructure for stimuli generation and associated checking and debugging of design behavior.

The effectiveness of input stimuli to exercise different execution scenarios in the design is of paramount importance to simulation-based validation. A metric to estimate the percentage of useful execution scenarios (according to a user-defined notion of useful) exercised by simulation vs. the total number of such scenarios is known as coverage metric. The goal of simulation-based validation is to reach the highest value of coverage possible in a given timeframe. An effective stimuli generation method is random generation constrained by design-specific restrictions on input validity, known as constrained random generation. Such generators are generally implemented in software. Hence, constrained random generators can be easily interfaced with a software-based simulator to exercise the design and maintain scoreboards to update the coverage metric.

All internal signals can be recorded during simulation and the verification engineer can easily debug the design by analyzing the simulation traces. Various software-based checkers are developed that can interface with simulator to perform checking, often in lockstep with the simulation itself. Assertions checking different aspects of correct behavior of the design are a commonly used verification construct, which are embedded in the design. Not all assertions can be represented efficiently in digital logic. However, all assertion-based checkers to detect and localize functional bugs can be co-simulated in behavioral fashion in software-based simulation. All in all, there is a rich set of verification solutions for this platform.

Software-based simulation is a fairly low-cost solution compared to hardware-accelerated simulation platforms, since the only cost is the software license and commodity hardware to execute this software. However, as discussed earlier, the speed of software-based simulation is not adequate for the growing verification needs of modern designs. A typical software-based simulator only achieves 1-10 simulation cycles per second when applied to a full-chip design. Such simulation performance renders any regression test longer than a few hundreds of thousands of cycles practically infeasible. The industry has already reached design sizes that are too large to be simulated in full detail at tolerable simulation performance with software-based simulation. For example, the full-chip simulation of the recent Intel Larrabee many-core design [84] entailed a memory footprint that was too large to handle by existing software simulators and as a result hardware-accelerated platforms were a basic necessity.

Clearly, any improvement in software-based simulation performance, at little additional

cost, enhances the ability of the existing methodologies. Hence, there is always a demand for low-cost and yet high-performance simulation software solutions. Any solution to improve simulation performance at a low cost can have a direct positive impact on verification performance and cost, while expediting the whole design cycle.

### 2.1.2 Acceleration platform

Acceleration platforms are composed of large arrays of customized ASIC processors, specifically designed to simulate logic gates concurrently. To target these platforms, a DUV must be synthesized into a structural netlist, and then the structural logic primitives are mapped to the execution substrate. Human effort to map a design to an acceleration platform is minimal, since essentially a compiler maps the logic for functional execution on special purpose logic processors and no physical logic is involved to create timing/electrical issues. Cadence palladium [23], IBM AWAN [30] are examples of such platforms. These platforms are extremely costly to build or purchase, and often cost upwards of few hundreds of thousands of dollars [23]. Simulation performances of these platforms are in the order of 10 kHz to 1MHz.

Acceleration platforms may experience performance penalties when increasing design size. Note that acceleration platforms do not have physical capacity limits similar to emulation platforms, since arbitrarily large logic descriptions can be simulated in a time-multiplexed serial fashion (different parts of the design being simulated sequentially for the same simulation cycle); however, the simulation performance can drop down to the level of software-based simulators after a certain degree of serialization.

Generally, acceleration platforms are attached to a host computer from which the simulation process is controlled and to which the recorded data is transferred. In current industry practices, the testbench is stored and executed on the host computer and controls the simulation running remotely on the platform. Selected signals are logged on the platform itself and periodically off-loaded to the host, where they are checked by a number of host-bound software checkers to establish the functional correctness of the simulated design. Transfer bandwidth to and from the host can be much smaller than that to support the transferring of data generated for the target checking activity. Hence, often, the logging and off-loading activities become the performance bottleneck of the entire simulation [67, 56].

Acceleration platforms allow the collection and transfer of signal values for debugging purposes, but the transfer slows down the simulation, eroding the key benefit of acceleration. In general, simulation performance is greatly reduced with increasing number of recorded signals. Performance losses of as large as 50% are reported for a recording rate of

only 100 bits/cycle [27]. Hence, even though these platforms offer a simulation speed in the kilohertz range, recording a large number of signals can bring it down in the sub-kilohertz range: the territory of software-based simulators. The method of collecting a subset of signals for recording is known as "tracing" and those signals are known as "traced signals". However, the precise relation between the number of traced signals and their impact to acceleration performance depends on the architecture of the acceleration platform. Reducing the number of recorded signals per cycle (thus the trace data generation rate) is extremely important to attain a successful checking solution for acceleration platforms. This is due to the fact that the underlying architecture of the acceleration platform records the values of the signals marked for observation in each cycle and stores them in internal memory; it must stop simulation every time the memory becomes full, transfer the content via a low bandwidth channel to a connected host machine and then resume simulation. The more frequently this event takes place, the higher the associated performance penalty. Thus, the lower the number of traced bits, the longer it takes to exhaust the internal memory resources, and the longer the intervals of uninterrupted simulation and higher the average simulation performance.

### 2.1.3 Emulation platform

Emulation platforms typically consist of programmable look-up tables arranged as a 2-dimensional array with programmable interconnect, known as field programmable gate arrays (FPGA) as a whole. Any digital system can be mapped on to such a platform by mapping the logic into a collection of lookup tables and attached memory modules. Note that a key difference between acceleration and emulation platforms is that, in the latter case, the whole system must be mapped and it executes system software in the same fashion as the final chip, instead of just mapping a portion of a system and running testbenches (which are traditionally on a separate host) to mimic the rest of the system. A design can be emulated at the clock frequency dictated by the constraints of the mapped logic, and clock speeds ranging from 10 MHz to 100 MHz can be reached. Although emulation attains higher performance than acceleration, the engineering effort needed to map a design into the platform is also very high since actual physical issues of timing / driving strength *etc.* are involved. For instance, BEE3 is an example of an emulation platform [32] that uses multiple Xilinx FPGA's. It is also interesting to note that there are acceleration platforms which use FPGA's as logic processors, as well as there are emulation platforms that are based on ASIC processors as the execution fabric. Emulation platforms may cost tens of thousands of dollars and this is further exacerbated by the engineering costs of mapping a

design to them.

However, from a checking and debugging point of view, the challenges are similar to that for the acceleration platforms. The number of lookup tables in the FPGA limits the amount of logic that can be fitted in these platforms, thus establishing strict limits on the logic capacity. Observability of internal signals is considerably lower compared to that of the acceleration platforms and requires embedded trace-buffers [95, 6]. Frequent transfer of simulation data to the host platform degrades emulation performance. As a result, emulation platforms have similar trade-offs from the standpoint of validation effort.

### 2.1.4 Silicon prototype

Silicon prototypes are early silicon versions of the design under verification. Execution speeds upwards of a Gigahertz can be obtained on these prototypes. Verification/debugging on such prototypes are known as post-silicon verification/silicon debug. Each iteration of fabricating these prototypes (known as a re-spin) is an extremely costly process (millions of dollars) and can incur delays of several months.

Silicon prototypes offer maximum performance but signal observability is at minimum. The capabilities of physical probing tools [72] are very limited, and it is infeasible to observe each and every signal in fabricated silicon. Often design for test (DFT) features, such as scan chains, are used for providing observability and debugging functional problems [92]. Though scan chains can capture all, or a subset, of internal state elements, and thus increase signal observability for silicon debug, it may take several thousand clock cycles to dump out one observed state snapshot and, in most cases, the circuit's execution must be suspended until the completion of this process. Hence, during the design phase, several dedicated design for debug (DFD) features [3] are also developed, which are utilized during the post-silicon verification phase. A common DFD feature is an embedded logic analyzer (ELA), which typically consists of a mix of trigger units and sampling units. Programmable trigger units are used to specify an event for triggering the logging of internal signal values, while sampling units are used to log the values of a small set of signals (traced signals) over a specified number of clock cycles into on-chip buffers known as trace buffers. However, since these structures provide no benefit to the final customer, the amount of silicon area that can be invested in them must be extremely small.

## 2.2 State-of-the-art in high-performance simulation-based validation

In an ideal world, a verification team would have a single platform that offers the strong checking and debugging capabilities of traditional software-based simulation, while providing the performance of acceleration and emulation platforms. However, as we move towards higher performance simulation solutions, the ease of checking and debugging is lost. One of the reasons behind this is the fact that, with the existing technology beyond software-based simulation, only structural logic descriptions can be simulated, emulated or fabricated. Lockstep execution of software-based checkers with hardware-accelerated simulation can be too detrimental to performance. One possible approach is to design checkers that can be synthesized as hardware and simulated alongside the design. If that is not possible – as it is often the case for complex checkers – an alternative viable approach is to trace and record relevant signal information during simulation, so that the checker can be run in a decoupled fashion. For example, a checker that validates the memory consistency model of a multi-core processor design can be too complex to be implemented entirely in hardware and a decoupled software checker that operates on the log of load/store and coherence messages would be the only feasible solution. Note that such tracing may require some additional tracing logic to be simulated alongside the design as well. Both approaches have been considered by researchers working in this field. We overview them below and discuss them in depth in Sections 2.2.1 and 2.2.2.

**Checker synthesis** is prone to run into logic footprint issues. The logic capacity related constraints associated with hardware-accelerated platforms prohibits the mapping of any arbitrary checking solution into equivalent hardware. Such translation can result in large checker logic overhead, which can erode the performance advantage of the platform. Thus, only checkers that result in low logic overhead can be tolerated.

**Log signals and then check** approaches have two types of associated overhead. First type is the performance penalty for tracing logic overhead and second type is the performance penalty for recording signal values. Hardware structures dedicated for tracing have certain logic overhead, which is most costly in the post-silicon phase since these additional structures do not have any purpose beyond debug, yet occupy valuable chip real estate. Apart from logic overhead, tracing signals also have detrimental effect on performance in acceleration and platforms as explained in the earlier section. Due to these effects, only a very small subset of signals can be traced in these platforms.

However, debugging often requires observing a much larger number of signals, which is only available during software-based simulation. Scarcity of observability is a common

fundamental problem that plagues platforms beyond software-based simulation. Hence researchers have explored ways to leverage partial signal information towards debugging, often resorting to approaches that attempt to **reconstruct non-observed signals**. Some of these approaches are discussed in Section 2.2.3.

Another approach to solving the debugging problem leverages the hardware-accelerated platform to reach a deep error state, snapshot the state, and then uses software-based simulation to **replay the interval of interest**. Debugging is relatively easy in this approach if the root-cause of the problem is contained within a small number of cycles from the error state, since we have full visibility into the design for software-based simulation. This approach is discussed in Section 2.2.4.

The limitations of each approach are discussed and the common set of challenges that are present today in validation beyond software-based simulation are delineated in Section 2.3. The goal of my dissertation is to overcome these challenges through innovative solutions.

### 2.2.1 Synthesizing checking constructs

A typical assertion checker constitutes of checking whether a certain sequence of events take place after the assertion is triggered by a certain type of event. This construct can simply be represented as a finite state machine (FSM) where state transitions are triggered by these aforementioned events. Such a FSM representation can then be synthesized into an equivalent sequential logic description. There has been a body of work by Boule, *et al.* [18, 20] to generate assertion checkers in form of synthesized hardware for acceleration platforms, emulation or silicon debug. Reconfigurable embedded checkers [3, 77] have been proposed for post-silicon validation as well. Acceleration and emulation platforms have constraints on logic capacity; hence not all checking solutions can be accommodated as synthesized hardware. Also the larger the amount of simulated logic, the worse is the simulation performance. These restrictions severely limit the checking capability on these platforms. So far only simple assertion checkers has been considered for synthesis, since the existing techniques do not extend to complex software checkers. As a result low hardware overhead checking solutions have become a necessity.

### 2.2.2 Tracing signals for off-line checking

A commonly deployed approach is to trace relevant signal data for checking, with the aid of debugging hardware (e.g. embedded logic analyzers) that is simulated, emulated or

fabricated with the design. The traced information is stored in trace buffers as explained earlier. Embedded logic analyzers have become common place for FPGA platforms [95, 6] and are used for ASIC as well [7]. Due to high speed of simulation, a large amount of data can be generated in the hardware in short duration, and since the trace buffers are small, frequent transfers are necessary. The larger the amount of recorded information per cycle, the more it degrades the simulation performance. Hence there is a growing need of recording information in a concise and compact manner during hardware-accelerated simulation or post-silicon tests, which later can be post-processed to gain deeper debugging insight. IFRA [76] is an example of such a solution implemented for post-silicon debug. Specifically, this solution records instruction's footprints while they traverse an out-of-order pipeline, which can later be utilized to detect a functional bug in the design. Generally the transfer bandwidths of acceleration and emulation platforms are fairly low compared to generation rate; hence transfer often becomes the bottleneck in this methodology, and the effective validation performance drops sharply. Also, these solutions often need knowledge of the architecture or micro-architecture of the design under verification.

### 2.2.3 Observability via reconstruction

As we move towards platforms with higher simulation performance, observing internal signals becomes more difficult. This is the most fundamental challenge in debugging: how to debug in presence of only partial knowledge of the design's behavior. It remains an active area of research, and only a few solutions have been proposed so far in this space. An example of a solution targeting post-silicon validation is BackSpace, proposed by DePaula, *et al.* [33]. A snapshot of the design state (which can be obtained via scan chains) is recorded when the error manifests and a small number of signals are traced for a number of clock cycles before the error state. From this information and by applying formal backward reachability analysis, it is possible to fully or partially infer the design state for a number of clock cycles preceding the error. This information can facilitate diagnosis of functional bugs.

Another approach to infer non-observed values from a set of observed values is state restoration, as first proposed by Ko, *et al.* [58]. In this approach a small number of state elements are traced over a number of clock cycles using a trace-buffer. Using this information and the logic representation of the circuit, a number of other state element values can be reconstructed, which can in turn facilitate debugging.

### 2.2.4 Replay from state snapshot

Replay from state snapshots is a solution where a design is simulated on a high-performance simulation platform, while snapshots of the design's state are taken at regular intervals and the input vectors are recorded. If the simulation process is deterministic, we can replay these short intervals of execution in a software-based simulator by loading the state snapshot into the design's flip-flops and then replaying the inputs for a particular interval. Since we are using the software-based simulator only for the short intervals, the length of the execution is well within the performance capacity of the simulator while it also offers complete observability of internal signals (being a software simulation). Synopsis Total Recall (TM) technology [89] is based on this approach, which is designed to work with emulation platforms where state snapshots can be obtained using scan chains and JTAG ports. However, this solution is only practical for medium-sized designs. In larger designs, obtaining complete state snapshots of a chip at a fairly high frequency, as needed for debug, becomes infeasible. Finally, the approach is not viable for non-repeatable simulation runs.

## 2.3 Key challenges

To summarize the discussion in the previous section, the challenges of simulation-based validation are:

**1. Attaining high-performance in software-based simulation:** Low-cost high-performance software-based simulation solutions are extremely valuable. If we can infuse performance into software-based simulation, its value in verification would greatly increase, since it already benefits from a high quality checking and debugging infrastructure. Thus high-performance always remains a priority.

**2. Providing signal observability in hardware-accelerated simulation platforms:** In general, reduced observability of internal signals hinders any debugging endeavor. Since we can only afford to trace a small number of signals, their selection becomes a crucial issue. We want to trace those signals that lead to inferring maximum amount of non-traced signal values, and thus provide the best possible observability into the DUV.

**3. Reducing logic footprint of embedded checkers in hardware-accelerated simulation platforms:** Low logic overhead hardware-embedded checkers are necessary as we transition more and more effort towards acceleration and emulation platforms. This is due to the

fact that these platforms have limited logic capacity and simulation performance degrades when increasing the logic footprint of the simulation.

**4. Compacting traced data:** Many checkers available in software-based platforms can be mapped to a practical embedded checker. In those cases, tracing of on-chip data for post-processing becomes necessary. However, simulation performance degrades with number of recorded signals. To retain simulation performance we must compress the relevant information for checking/debugging so that a small number of signals are traced and yet we can gain deep insights on the design activity, even when tracing a few signal values.

**5. Developing a methodology for effective validation in hardware-based platforms:** The final challenge is to develop a methodology that allows us to adapt the checking solutions available in software-based simulation platforms to hardware-accelerated platforms in a general fashion.

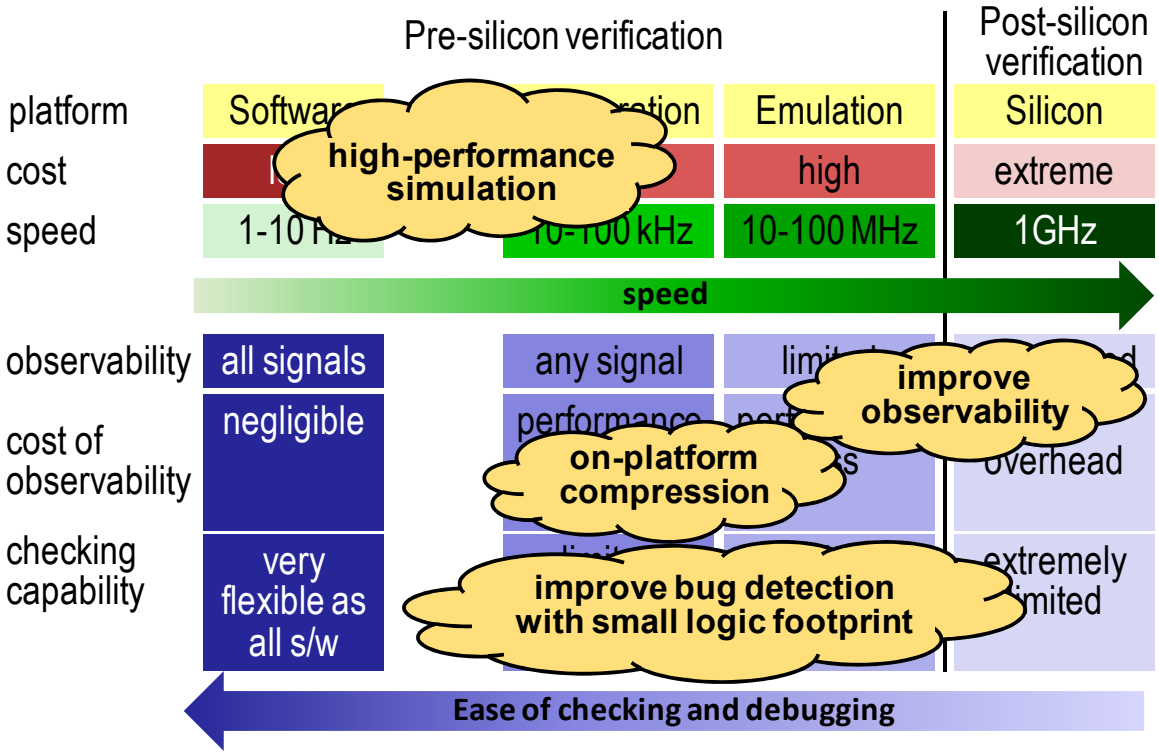



**Figure 2.2 Challenges and scope of research** to enable effective high-performance simulation-based validation.

These research challenges are also summarized in Figure 2.2, organized according to their scope in relation to the characteristics of the platforms. The goal of this dissertation is to provide solutions to tackle these challenges. The solutions described in the dissertation

are targeted towards the common set of challenges; however specific implementations only target specific platforms.

## 2.4 Contributions

The rest of this dissertation describes the solutions contributed to tackle the challenges mentioned above. This section introduces outlines them. The first challenge of attaining low-cost high-performance software-based simulation is solved by introducing simulation software that can exploit the massive parallelism available in modern hardware in the form of graphics processing units (GPU). The challenge of providing signal observability for platforms beyond software-based simulation is tackled by a signal restoration algorithm inferring values from traced signals. Section 2.4.2 introduces a solution that provides improved signal observability by enabling selection of trace signals with high restoration potential. The next three challenges of reducing logic footprint, compacting traced data, and providing a general checking methodology for hardware-accelerated simulation are tackled in the context of microprocessor designs and are outlined in Section 2.4.3. Note that the key ideas developed in that Section are applicable to other classes of designs as well. The challenge of logic footprint reduction is countered by a novel technique of checker approximation; trace data compaction is achieved by performing on-platform compression and adapting off-line software checkers to operate on compressed data. Finally, a methodology for adapting checkers developed for software-based simulation to hardware-accelerated platforms is introduced by combining embedded checkers and off-line checking on compressed data in a hybrid fashion.

### 2.4.1 Infusing performance into software-based simulation

One of the major contributions in this dissertation is in delivering high-performance software-based logic simulation at a low cost. Logic simulation is used to verify designs at the behavioral level, as well as the structural level, ensuring that a synthesized circuit's netlist matches the functionality and timing of the behavioral model. Structural netlists are particularly cumbersome for simulation because of their low-level specification and the fine granularity of the structural definition, which consists of large number of gate primitives in the target technology library. The recent availability of general purpose computing programming models for high-performance and highly parallel GPUs (GP-GPUs) led us to explore a new simulation architecture targeting these hardware platforms, with the hope of

delivering a conspicuous performance advantage at a small hardware cost (that of a GPU peripheral). The resultant simulation solution, called GCS, exposes the parallelism available in the simulation problem to the parallel processing hardware of the GPU, by using novel partitioning algorithms.

Similar partitioning schemes are found to be applicable for simulation of behavioral descriptions as well. This leads to a high-performance simulation solution called SAGA for behavioral design descriptions expressed with the synthesizable subset of SystemC. Since, all these simulation solutions are software-based, existing checking and debugging solutions can be adapted with minor software engineering effort. Chapter 3 describes these solutions in detail.

### 2.4.2 Providing observability through restoration

In this research thrust, the dissertation presents a solution to achieve improved debugging capability in the wake of limited signal observability that is endemic to acceleration platforms, emulation and post-silicon validation. A solution to address this issue leverages trace buffers: these are register buffers embedded into the design with the goal of recording the value of a small number of state elements, over a time interval, triggered by a user-specified event. Due to the trace buffer's area overhead, designers can afford to trace only a very small fraction of a design's signals. Clearly not all signals are equally useful for debugging and diagnosis. Thus, effective trace signal selection is critical towards the success of debug.

Recently, researchers have demonstrated that observability can be provided via reconstruction of non-recorded signal values from the recorded signal values [58]. Such observability is critical for debug. Ideally, we would like to select signals enabling the maximum amount of reconstruction of internal signal values. A novel selection algorithm, aided with an accurate restoration capacity metric is presented in this dissertation. This solution overcomes some of the key shortcomings of previous signal selection algorithms, and leads to attaining a higher degree of restoration than previous solutions. Since this methodology does not require any design specific knowledge it is applicable to any logic block. The details of this solution are described in Chapter 4.

### 2.4.3 Enabling checking capability in hardware-accelerated platforms

The rest of the contributions are geared towards bringing in checking and debugging solutions, which are currently only feasible within software-based simulation, to the realm

of hardware-accelerated simulation. These solutions require a thorough understanding of the design under verification and have been developed targeting micro-processor designs. However, the underlying principles apply to other classes of designs as well. Acceleration platforms were the primary target for these solutions, but the key ideas are equally relevant to emulation, or even post-silicon platforms.

### Embedded low logic footprint approximate checkers

The first approach is an attempt to bring in existing software-based checkers that are used with software-based simulation, into the purview of accelerated simulation. As discussed before, these platforms do not provide the rich checking capabilities of software-based simulation methodologies. As a result, mapping checkers, particularly complex checkers, such as golden models or checkers making use of complex software data structures, remains a challenge because (i) embedded checkers can only use synthesizable constructs, (ii) their logic complexity should not exceed the platform capacity and (iii) the performance impact entailed by the simulation of their logic components should not be such to make the acceleration performance comparable to that of a traditional software-based simulation.

This dissertation describes a novel solution to bring in those complex checkers, typical of software-based simulation environments, onto acceleration platforms. To this end, checkers must be transformed into synthesizable, compact logic blocks; yet, they should have bug-detection capabilities similar to that of their software counterparts. "Approximate checkers" trade off logic complexity with bug detection accuracy by leveraging novel techniques to map complex software checkers into small synthesizable hardware blocks, which can be simulated along with the design on an acceleration platform. In Chapter 5, a general checker classification is presented; a range of approximation techniques, based on the characteristic of the checker, is proposed; and finally, appropriate metrics for their evaluation are presented.

### Using on-platform traced data compression

Checker approximation is followed by another checker adaptation approach. A case-study is presented for a microprocessor checking component that is complex enough that it cannot be translated to hardware. As a result a "log-and-then-check" approach is necessary to map this type of checkers. For this situation, the dissertation proposes event tracing by additional simulated logic followed by post-simulation checking. As discussed before since only very few signals can be recorded without degrading the acceleration performance, the

methodology itself needs to be adapted to operate with compressed or partial information.

This concept is demonstrated on a common checking solution for microprocessor validation: namely instruction by instruction checking (IBI). IBI checking tracks the architectural events generated by a microprocessor design model when it is executing a test regression and compares them against a golden architectural model. However, if all data associated with these architectural events were traced, it would severely degrade the performance of the acceleration platform. Hence, it is imperative to produce a summary of the information needed for checking, using only a few bits of information collected per cycle. In the proposed novel scheme, a summary of the events are produced by additional simulated hardware, and the checker is adapted to operate on checksums, instead of actual architectural register values. This approach results in a checker that is almost as accurate as software-based solution, yet offers the same performance as that of the hardware-accelerated platform. This solution is also discussed in Chapter 5.

### Hybrid checking

Finally, hybrid checking leverages light-weight embedded checkers, on-platform compression as well as a post-simulation checking component that operates on the compressed simulation trace. This solution attempts to combine the beneficial effects of both approaches and suggests a comprehensive methodology for adapting complex checkers into the realm of hardware-accelerated simulation. The methodology involves classifying typical microarchitectural checks needed for a modern microprocessor design into cycle-accurate local embedded assertions (implemented as lightweight embedded logic) and event-accurate functionality checks requiring a post-simulation checking phase. Embedded logic is further used to compress the data associated with events relevant to functionality checks. This solution is discussed in Chapter 6.

# Chapter 3

# The Quest for Simulation Speed

As explored in the previous chapters, the majority of validation methodologies in the industry rely heavily on the use of design simulation platforms. These platforms simulate a design's functional behavior at different levels of abstraction, ranging from high-level behavioral to low-level structural gate-level descriptions. The primary platforms for simulation-based validation are software-based simulators executing on general-purpose computers. The increasing complexity of modern designs has been pushing the scalability limits of software-based simulation: as of today its poor performance on complex designs has heavy impact on the development timeline and ultimately on a product's time-to-market [36]. Currently, the performance of such software-based simulators are not even close to adequate to meet the validation demand. The performance limitation is intrinsically tied to the single threaded nature of such simulators targeted towards conventional general purpose processors as the execution substrate. However, to its credit software-based simulators offer excellent checking and debugging support, hence infusing performance into software-based simulation will be extremely beneficial for validation. This chapter explores the potential of increasing the performance of software-based simulation by exposing the parallelism available in the problem to the massive parallelism available in graphics processors.

## 3.1 High-performance simulation through massive parallel processing

**Software-based simulation** of a design's description remains the primary methodology of validation in the industry. In this methodology, a software simulator application executes on a general-purpose computing machine (such as a desktop or a server), reads in files describing the design and the associated testbench, and then simulates the design as intended. Most of the design checking and debugging tools, such as assertion-based infrastructures and simulation trace visualization tools, are connected on top of the software-based simula-

tor. The deployment of a **complex checking and debugging infrastructure comes fairly easy** in this environment, as these components can be connected to the simulator's software via simple programming interfaces. Verification engineers in the design houses attempt to simulate as many simulation cycles as possible before final design tapeout: to attain higher degree of coverage and to detect and remedy as many functional bugs as possible, with the assistance of the available checking and debugging infrastructure.

Unfortunately, software-based simulation performance falls short of expectation even in the face of decades of improvements in the performance of these tools by the EDA industry. They still lack the horsepower required to tackle today's complex digital designs. A full-chip simulation for a moderately sized design, only runs at the speed of 1-10 simulation cycles per second on a software-based simulator, thus severely restricting exploration of the state space of such a design via software simulation. Moreover, a large design might not even fit in the memory available in a general-purpose machine. In the industry often a number of simulations are performed on the same design with different initial conditions and different input stimuli, which can execute in parallel in different servers, and can explore different portions of the design state space. However, this does not solve the fundamental problem of reaching deep into the state space in a feasible amount of simulation time. Functional bugs can hide deep into the state space where a short depth simulation run would not be able to reach. In purview of the comprehensive checking and debugging support available in software simulation, it will be ideal if we could **increase the performance of the simulation process without perturbing the essential software nature of the simulator**.

An investigation into the performance bottlenecks of current software-based simulators reveals that the performance limitation is intrinsically tied to the single threaded nature of the simulator design targeted towards conventional general purpose processors as the execution substrate. However, the **simulation process is inherently parallel in nature**, since at any abstraction level, there are multiple components of the design that can be simulated in parallel. As a result, traditionally used general purpose processors have a fundamental limitation due to the fact that **simulation is forced to be serialized** at some granularity due to the very nature of the execution substrate. Even though general purpose processor performance have increased over past few decades due to Moore's law scaling, it has reached a point of stagnation where single thread performance has very little improvement over generations of processors, while the number of processor cores available in a compute unit has increased. Modern general purpose processors can support multiple parallel threads, executing in different cores. Clearly **simulation can benefit from parallel execution**, and it often possesses a far higher degree of parallelism than a few threads.

Naturally, software-based simulation performance can be heavily boosted by leveraging a massively parallel execution substrate. Graphics processors(GPU) already possess **massive execution parallelism** since they have to maintain a massive rate of pixel throughput for rendering graphics. Specifically **general purpose graphics processing units (GP-GPU)** are a special class of processors where the massively parallel computation capability in the hardware can be accessed via a high level language for general purpose computation. Hence, this execution substrate is a perfect fit for the simulation problem. Moreover, GP-GPUs are a cheap off-the-shelf commodity, thus they are very well suited for accelerating software-based simulation at nominal additional hardware cost.

### 3.1.1 Overview of this chapter

Parallel processing can accelerate simulation at different abstraction levels. In the scope of this chapter, two different design abstraction levels are targeted in particular, i) structural gate-level representation and ii) a subset of behavioral descriptions expressed in SystemC language. Gate-level simulation is essential since it is often necessary to validate equivalence of a structural model of the design against a behavioral model. The performance of software-based simulation on general purpose processors is excruciatingly poor when simulating gate-level netlists, where the system's description is fairly detailed, leading to a large design model. On the other hand SystemC is widely used in the full system design exploration process in the industry and is vital for hardware software co-development, while the simulation performance of SystemC is far from adequate. This dissertation explores the possibility of accelerating software-based simulation performance of digital designs at these two abstraction levels using GP-GPUs as execution substrate. To this end a logic simulator called **GCS (GPU concurrent simulator)** for functional simulation of gate-level netlist is developed. GCS was able to deliver an order of magnitude performance improvement over software simulators on general purpose processors. On the other hand I collaborated in development of a parallel SystemC simulation framework called **SAGA (SystemC Acceleration on GPU Architectures)** that leverages GP-GPUs. This simulator was also able to deliver up to an order of magnitude performance improvement over traditional solutions.

GP-GPUs offer massive parallelism, however they have strict restrictions regarding the nature of the workload, and offer the best performance when presented with a very regular execution pattern and a partitioning of the workload into independent groups of worker threads. Even though both of the simulation problems have a large degree of parallelism, it was crucial to properly partition each of these problems to best match the parallelism

present in the execution substrate and methods to regularize the execution pattern was critical as well. To this end I developed novel algorithms to perform partitioning of the computation best suited to GP-GPU paradigm, as well as novel computation morphing methods to regularize the execution pattern as much as possible under the constraints of the problem. To fully comprehend the nuances of adopting simulation on the GP-GPU platform, we require a basic understanding of the GP-GPU architecture and the programming model. Hence, a brief overview of GP-GPU architecture and programming abstraction in Section 3.2 is presented first, followed by the details of the GCS solution through Sections 3.2 to 3.6, which comes in both oblivious and event-driven flavor. Then, the SAGA solution is presented in detail through Sections 3.7 to 3.9. An account of prior literature relevant to this research is presented in Section 3.10 before concluding this chapter.

## 3.2 Introduction to GP-GPU architecture and programming model

General purpose computing on GPUs enables parallel processing on commodity hardware. Since 2007, NVIDIA has provided a programmer-friendly approach towards GP-GPU computing, aiming at facilitating parallel programming with a new general purpose programming interface and architecture known as NVIDIA's Compute Unified Device Architecture (CUDA) [73]. There are other programming interfaces for GP-GPU computing available as well such as OpenCL [54] from the Khronos initiative. We chose CUDA as an example programming model since it encompasses the key features of any GP-GPU programming interface. In the CUDA execution model, the GPU is a co-processor capable of executing many threads in parallel, following the single instruction multiple data (SIMT) model of execution. A data parallel computation process, known as a kernel, can be offloaded to the GPU for execution. This model of execution is known as single instruction multiple thread (SIMT), where thousands of threads execute the same code, each operating on different portions of data. The collection of threads represented by a kernel is divided into a grid of thread-blocks, each of which consists of a number of threads. Threads identify their spatial location within the kernel by thread block ID in the grid and then thread ID within the thread-block, and can use this information to access a different data location.

The CUDA architecture (Figure 3.1) consists of a number of multiprocessors (up to 16 in the current generation) contained in a single GPU chip. Each multiprocessor is comprised of multiple stream processors (32 in current generation) which have common instruction fetch and support a large number of concurrent threads (up to 1024 in the current

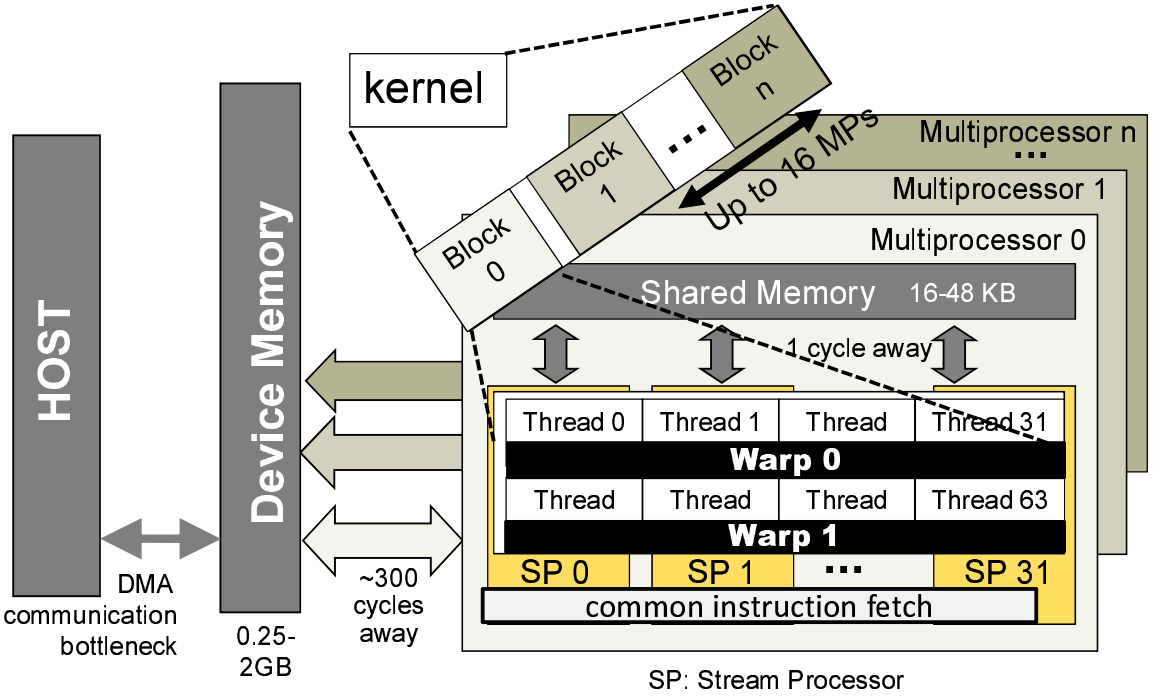



**Figure 3.1 NVIDIA CUDA GP-GPU architecture.** A GPU includes a number of multiprocessors, each comprising 8 stream processors. Several threads (up to 512) may execute concurrently within a multiprocessor and communicate through a small shared memory bank. The larger device memory has much higher access latency.

generation) all running the same code. Multiprocessors are responsible for the execution of the thread-blocks that can be mapped to each of them, as dictated by resource limits. Each multiprocessor has access to low latency (1 clock cycle) scratchpad memory, divided between local registers and shared memory. A thread-block has exclusive access to a portion of this scratchpad memory, meant for collaborative use between threads of a particular thread-block, which is not accessible by other thread-blocks. All multiprocessors also have access to a region of global memory called device memory, which has higher access latency (300-400 cycles) while the capacity can be 256 MB to 1 GB in current CUDA enabled GPU's. While the access latency to global memory is high, it is possible to amortize the cost by coalescing accesses from multiple threads. Communication with the host CPU's main memory is achieved by means of direct memory access (DMA) transfers, which are most efficiently performed for large blocks. For best performance, it is important to keep communication between the host and the GPU to the bare minimum, e.g., by copying all relevant data-structures to the device memory and not communicating with the host memory during execution at all, and copying back the final results. A thread-block can allocate a certain amount of shared memory dedicated for co-operative usage by it's threads and will also take up a certain number of local registers as dictated by the code in the body of each

individual thread. Each multiprocessor will be responsible for the execution of the number of thread-blocks that can be maximally contained in it, as dictated by resource limits.

Since all resident threads in a multiprocessor execute on the fixed number of stream processors (32 in the current generation) with a common instruction fetch unit, each thread-block executes groups of 32 threads at a time (known as a *warp*) in a time-multiplexed fashion, with frequent context-switches from one warp to another, happening on regular intervals or long latency global memory accesses. Because of the shared fetch unit, execution path divergence between threads of a same multiprocessor is detrimental to performance as only one branch path can be executed at a time. If threads in a same multiprocessors must execute different code paths, the least penalizing solution is to map them to different warps, so that the memory accesses originating from different warps can be partially overlapped in time. Threads belonging to a single thread-block can be synchronized using fast barriers, while synchronization across multiprocessors can only be achieved via kernel termination. While designing any software application for GP-GPUs we have to take in to account all these GPU-specific constraints.

## 3.3 Towards high-performance logic simulation

In a typical digital design flow, a system is first described in a high-level behavioral fashion with a hardware description language (HDL), then it is automatically synthesized to a netlist, consisting of structural logic elements such as logic gates and flip-flops. To ensure that the gate-level design provides the same functionality as the behavioral design, the former must be validated by thorough simulation and comparison with the behavioral model. These structural netlists can easily be comprised of tens of millions of gates in modern digital systems. A logic simulator takes this netlist as input, converting it to internal data structures: feedback loops are opened by disconnecting the sequential storage elements in the design, thus allowing to simulate the design one cycle at a time, storing the value of latches and flip-flops in internal data structures of the simulator software. The remaining logic, that is, the combinational portion, is then levelized according to the dependencies implied by the gates input-ouput connections. Simulation proper can now begin: the simulator generated input values and then computes the outputs of the internal logic gates, one level at a time, until the design's output values are produced. In subsequent simulation cycles, the values computed for the design's storage elements are looped-back and used as part of the next cycle's inputs.

Logic simulators comes in two flavors: oblivious and event-driven. In an oblivious sim-

ulator, the simpler of the two simulator flavors, all gates in the design are computed at every cycle. While the program's control flow for this approach is low-overhead, computing for every gate at every cycle can be time-consuming and, most importantly, unnecessary for all those gates whose inputs have not changed from the previous cycle. Event-driven simulation, on the other hand, takes advantage precisely of this fact: the output of a gate will not change unless its inputs have changed. Large portions of the design are often quiescent during a given simulation cycle, thus event-driven simulation can spare a large amount of redundant computation. Note, however, that the key to a successful event-driven simulation lies in the effective management of the additional program control overhead, necessary to track which gates must be re-computed and which are quiescent.

Structural gate-level simulation benefits from inherent parallelism as the logic corresponding to different outputs can be simulated in parallel. However, available commercial simulators, operate primarily on single threaded processors, thus they do not exploit this potential for concurrent computation available in the data structure representing the netlist. In this chapter, we investigate how the parallelism available in the problem structure can be mapped to that of the execution hardware of GP-GPUs. To this end, we use novel algorithmic solutions to address a netlist's structural irregularity, as well as techniques to exploit a GPU's memory locality in an optimal manner. While the parallelism of netlists matches well with the parallel computational power available in GPUs, there are a number of problems that must be addressed to enable GPU-based logic simulation. First, a netlist must be partitioned into portions that can be mapped and simulated concurrently and efficiently on a GPU. The partitioning must be aware of the GPU architecture and its memory model. Additionally, we need low-overhead algorithms to efficiently control the simulation of all design's portions.

In the following few sections, two novel simulator designs are described that leverage the parallel processing capabilities of low-cost general purpose graphics processing units (GP-GPUs) for gate-level simulation, leading to a major improvement in simulation performance. The first design is an oblivious simulator which utilizes simple scheduling, static data structures and better data locality. While the second is an evolved event-driven design which performs event-driven simulation of the netlist at a coarser granularity than individual gates, and requires dynamic analysis for scheduling re-evaluation. The recent availability of general purpose computing programming models for high-performance and highly parallel GPUs led us to explore a new simulation architecture targeting these hardware platforms, with the hope of delivering a conspicuous performance advantage at a small hardware cost (that of a GPU peripheral). Specifically, the NVIDIA's CUDA architecture provides a programming interface that enables users to develop software applications for

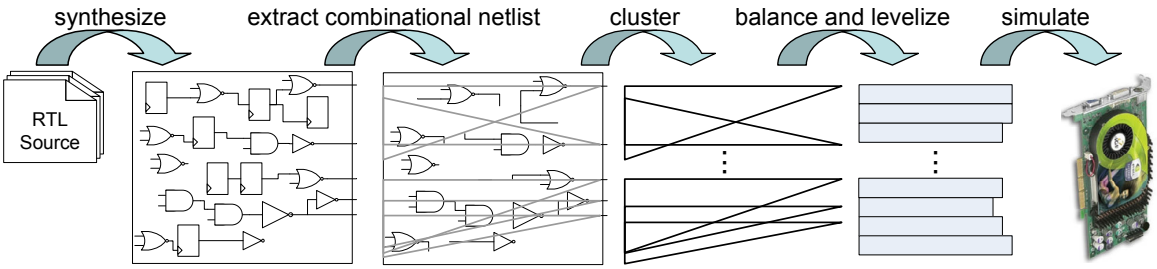



**Figure 3.2 The GCS compiler** considers a gate-level netlist or synthesizes a behavioral netlist. It then extracts the combinational logic block and partitions it into clusters, that is portions of the circuit that approximately fit within the resources of a single CUDA multiprocessor. The balancing step then optimizes each cluster to satisfy CUDA resource constraints. Finally, balanced clusters are transferred to the GP-GPU device and the simulation commences.

their vastly parallel co-processor GPU. However, CUDA exposes its parallel architecture directly to the programmer, with the result that applications must be designed specifically for this architecture in order to derive benefit from it.

## 3.4 Oblivious simulator overview

The oblivious GCS simulator operates as a compiled-code simulator, first performing a *compilation*, where it considers a gate-level netlist as input, compiles it and maps it into CUDA. A *simulation* proper follows, where GCS considers a CUDA-mapped design, simulating over a number of several cycles, possibly reusing the same mapped design while running with many distinct testbenches. The process of compilation and simulation progresses in 5 steps (Figure 3.2). First, a behavioral netlist is synthesized to a gate-level netlist and mapped to GCS's internal representation. From here, the combinational elements are extracted, since the design will be simulated in a cycle-based fashion. Next, GCS partitions the netlist into *clusters*, that is, logic blocks of appropriate size to fit within the constraints of the CUDA architecture. In this phase, the compiler prepares rough clusters, based on size estimates quickly computed on the fly. The following step, *balancing*, is an optimization phase, where each cluster is carefully restructured to maximize compute efficiency during simulation. Finally, all the required data structures are compiled into the CUDA kernel and transferred to the GP-GPU device. Testbenches can be implemented using many different solutions; if they are encoded in a CUDA program (possibly with associated stimuli data), then the simulation can be completely offloaded from the host with direct performance benefits. If the testbench resides on the host, control alternates between host and GPU to simulate and generate stimuli.

### 3.4.1 Synthesis and combinational netlist extraction

The GCS compiler requires a gate-level netlist as input. This can either be a synthesized version of a design under verification, or a behavioral description to which we can apply a relaxed synthesis step. In our experimental evaluation, we consider a broad range of designs, including a pool of behavioral descriptions that we synthesized using Synopsys Design Compiler targeting the GTECH library. Within the GTECH library we excluded non-clocked latches (but not flip-flops), since a cycle-based simulator cannot properly handle the sub-cycle delays involved in the simulation of a non-clocked latch. Multiple clock designs can still be handled by using a logical clock that generates all other clock signals. When the netlist is read into GCS, an internal representation based on GTECH is created. In GCS we represent each gate's functionality by a 4-valued (0,1,X,Z) truth table.

During the compilation phase, GCS extracts the combinational portion of the gate-level netlist and maps it to CUDA, creating data structures to represent the gates, as well as their input and outputs. During simulation, dedicated data structures store the simulated values for the storage elements (the *input* and *output buffer vectors*) and specialized testbench code feeds primary input values and extracts primary output values at each simulation cycle.

Because of the memory hierarchy of CUDA, an optimal memory layout can lead to significant improvements in the performance of a GP-GPU simulator. GCS places the most frequently accessed data structures in local shared memory (Figure 3.3). Here, we store intermediate net values (called the *value matrix*), which are computed for each internal netlist node during simulation. Also in local shared memory the *gate-type truth tables* are stored, which are consulted for the evaluation of each gate.

All other data structures reside in the higher-latency device memory: the *input* and *output buffers* and the *netlist topology* information. Note that the netlist topology information is required just as often as the data that we store in the local memory. However, the latter is data that is shared among several threads (gates) and thus its locality can benefit multiple threads.

### 3.4.2 Clustering

GCS's clustering algorithm (Figure 3.4) divides a netlist into clusters, each to be executed as a distinct thread block on the CUDA hardware. Since CUDA does not allow information transfer among thread blocks within a simulation cycle, all thread blocks must be independent. The central goals of the clustering algorithm are (i) minimizing redundant computation, (ii) data structure organization and (iii) maximizing data locality.

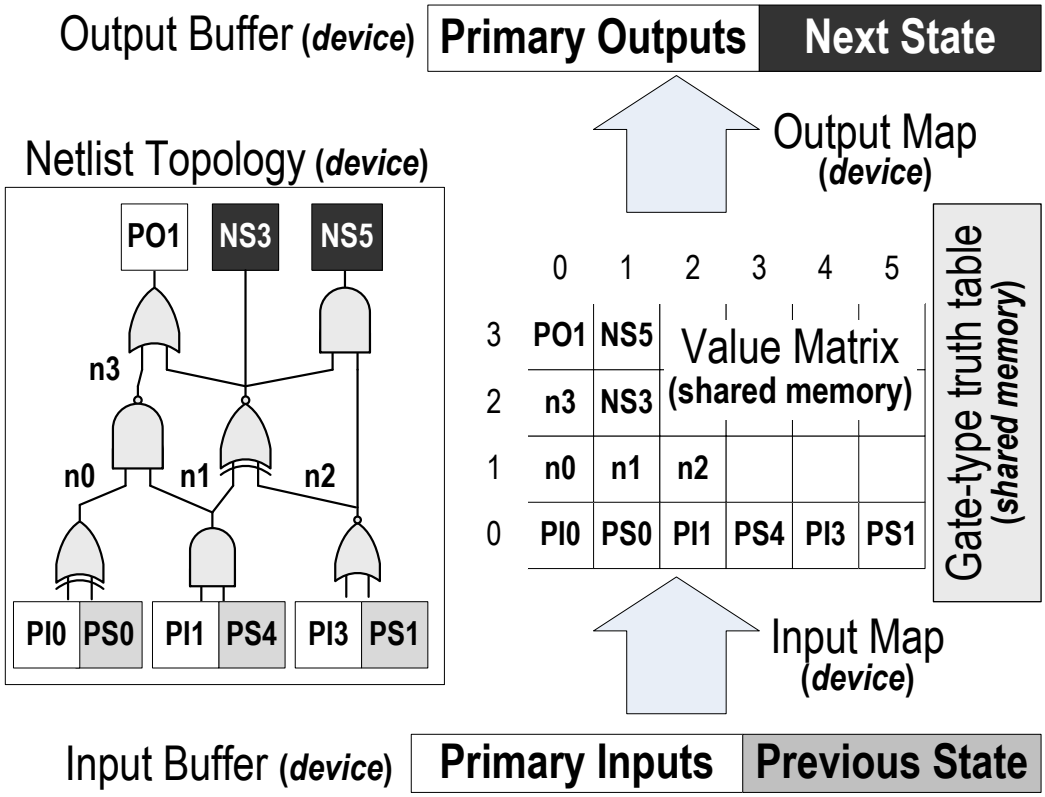



**Figure 3.3 GCS's compiled-netlist data structures**. The picture shows the data structures required for the simulation of a small netlist. Thread blocks store and retrieve intermediate net values from the value matrix in the local shared memory. Note that there is a one-to-one correspondence between a row of intermediate values and a netlist's logic level.

The requirement of creating netlist clusters that are self-contained and do not communicate to other clusters within a simulation cycle led us to choose a *cone partitioning* approach. In cone partitioning, a netlist is viewed as a set of logic cones, one for each of the netlist's outputs; each cone includes all the gates that contribute to the evaluation of that output. Due to the lack of inter-cluster communication capability, each cluster must include one or more cones of logic, and each cone must be fully contained within a cluster. Cone overlap necessarily requires that some gates are duplicated, because they belong to multiple cones. However, the incidence of this extra computation is small in practice.

During the simulation of a cluster, several data blocks must be readily available. Because each thread block has fast access only to the small local shared memory, the size of this structure becomes the constraining parameter in our clustering algorithm.

With the goal of minimizing cluster overlap, the clustering algorithm proceeds by assigning one cone of logic – we start from the one with the most gates – to a cluster. Additional cones are subsequently added to this cluster until memory resources have been exhausted. The criteria for adding a cone is the *maximal number of overlapping gates*; for

example, the second logic cone is the cone that overlaps the most with the first one already included in the cluster. Upon completion of the clustering algorithm, GCS has mapped all gates to a set of clusters, minimizing logic overlap while satisfying the constraints of shared memory resources.

```
1: sort(output_cones);
2: for each output_cone do
3:     new_cluster = output_cone;
4:     while size(cluster) < MAX_SIZE do
5:         cluster += max_overlap( output_cones, cluster );
6:     end while
7:     append(cluster, clusters);
8: end for
```

**Figure 3.4 Pseudo-code for the clustering algorithm.** Combinational logic cones are grouped into clusters, netlist blocks that are estimated to fulfill CUDA's resource constraints, with minimal logic overlap.

### 3.4.3 Cluster balancing

The *cluster balancing* algorithm minimizes the critical execution path of thread blocks (clusters) on the CUDA hardware. It considers each cluster individually and optimizes the scheduling of each gate simulation so that the number of logic levels (the limiting factor for execution speed) is minimized. The simulation latency of a single cycle is limited by the cluster with the most logic levels, since each additional level requires another access to device memory 300-400 cycles away. Considering the number of logic levels (cluster *height*) and the number of concurrent threads simulating distinct gates (cluster *width*), the algorithm balances these within the constraints of the CUDA architecture: a maximum of 256 concurrent threads. Since this is a functional simulaton, intra-cycle timing can be safely ignored and thus the transformation is guaranteed to generate equivalent simulation results.

### 3.4.4 Simulation

After the balancing step, the GCS compiler has generated a finite number of clusters, optimized them and generated all the support data structures necessary for the kernel code to simulate all gates in a netlist with a high level of parallelism while respecting data dependencies. At this point, cluster data and kernel code can be transferred to the GP-GPU

device and simulated cycle by cycle.

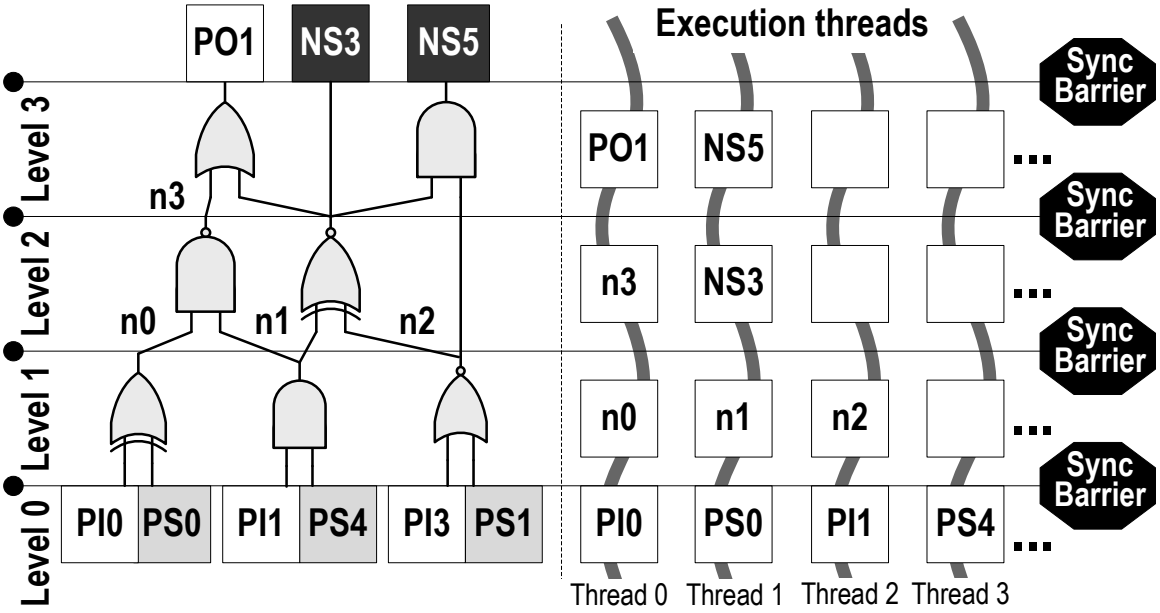



**Figure 3.5 GCS simulation on CUDA**. Simulation of the small netlist of Figure 3.3. Each thread is responsible for computing the output of one gate at a time, vertical waved lines connect the set of value matrix slots for which a single thread is responsible at subsequent time intervals. Note also how each level is followed by a synchronization.

Cluster execution on the GPU proceeds in three phases: *scattering*, *logic evaluation* and *gathering*. During scattering, the cluster's primary input data is retrieved from the device memory and copied to the value matrix (Figure 3.3). Next, logic evaluation progresses when each thread begins execution. The threads, each simulating one gate, retrieve the relevant portion of the netlist from device memory, as well as gate truth tables and net matrices from local shared memory. With this information, the threads evaluate their gates by consulting the truth table. During the gather step, computed results are copied from the value matrix to the output buffer vectors in device memory. Finally, the threads synchronize after simulating their respective gates and the process is repeated for all the subsequent logic levels in the cluster. Figure 3.5 shows an example of cluster execution for the sample netlist of Figure 3.3.

## 3.5 Event-driven simulator overview

The oblivious simulation solution of the previous section posseses a simple software design, and can be optimized statically, but simulating all gates in each cycle is redundant and also limits the performance of this approach. Moreover, the size of the circuits that can be simulated is severely limited by the size of the shared memory in the GPU platform. To

address these issues in order to achieve better performance for the common case, event-driven simulation is considered. However due to the particular architecture of GP-GPU, event-driven simulation at the fine granularity of gates would be inefficient, event-driven design can only be efficient at a much coarser granularity. The steps of logic synthesis and subsequent extraction of the combinational circuit remain the same as the oblivious simulator, however the compilation phase in this case is responsible for segmenting a large monolithic netlist into blocks amenable to simulation by individual execution units within the GPU. This requires segmenting the netlist into *macro-gates*: a set of several connected gates within the netlist of ideal size, optimizing the logic within each macro-gate, and finally producing the data structures and the CUDA programs necessary to carry out the simulation. During simulation, both program and data reside on the GPU. The testbenches are implemented in the same fashion as in the oblivious simulator, reading outputs and feeding inputs at the end of each clock cycle.

### 3.5.1 Segmentation into macro-gates

To exploit the advantage of event-driven simulation at a coarser granularity, we must segment the gate-level netlist into several logic blocks (called *macro-gates*), and assign the simulation of each macro-gate to a distinct CUDA multiprocessor. During simulation, we maintain a *sensitivity list* of nets at the inputs of each macro-gate: if any net in a sensitivity list changes value, then the corresponding macro-gate will be affected by the change and must be simulated (*i.e.activated*). Otherwise, the macro-gate can be skipped during the current cycle.

In determining how to partition the netlist into macro-gates, we took into consideration several factors: (i) the time required to simulate a macro-gate should be greater than overhead of determining which macro-gates to simulate; (ii) CUDA's multiprocessors can only communicate through device memory, thus macro-gates should not share data. To this end, we occasionally duplicate small portions of logic, so that each macro-gate can compute the value of its outputs independent of other concurrent macro-gates. Finally, (iii) we want to avoid cyclic dependencies between macro-gates, so to simulate each macro-gate at most once per cycle.

To address the list of constraints, we segment the netlist by partitioning the netlist into *layers*: each layer encompasses a fixed number of the netlist's levels. Macro-gates are then defined by selecting a set of nets at the top boundary of a layer, and including its cone of influence back to the input nets of the layer. The number of levels within each layer is called the *gap* and corresponds to the height of the macro-gate. By using this procedure,

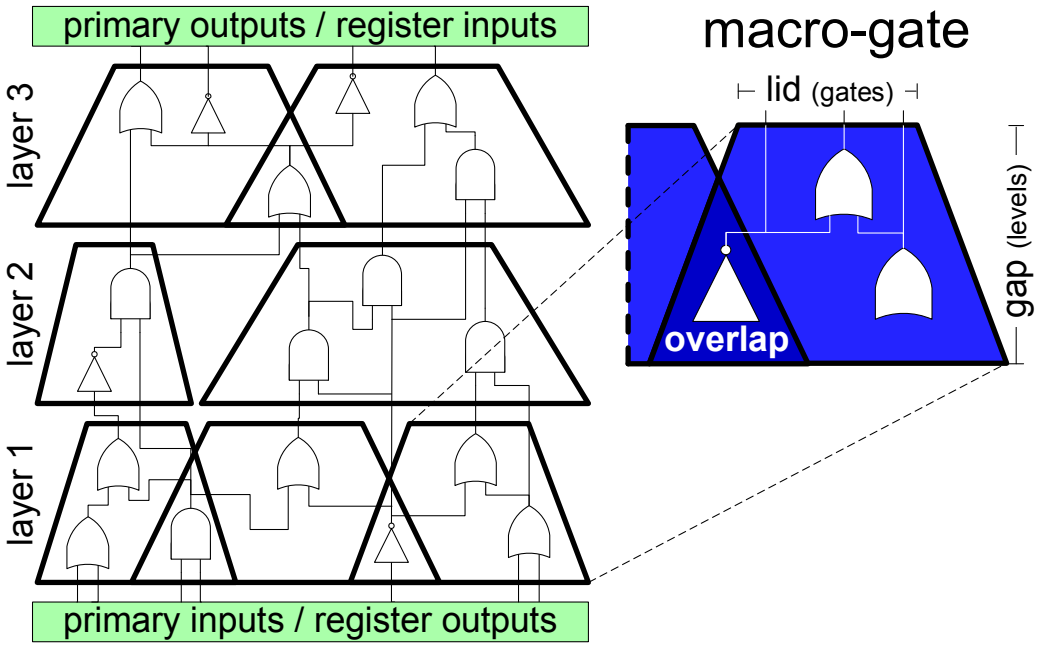



**Figure 3.6 Segmentation topology**. The levelized netlist is partitioned into layers, each encompassing a fixed number of levels (gap). Macro-gates are then carved out by extracting the transitive fanin from a set of nets (lid) at the output of a layer, back to the layer's input. If an overlap occurs, the gates involved are duplicated to all associated macro-gates.

it is possible that a given logic gate is assigned to two or more macro-gates. In this case, we duplicate it, so that each macro-gate can compute the value of its output nets without sharing any data with other macro-gates (second requirement). Finally the number of output nets used to generate each macro-gate is a variable parameter (called *lid*), whose value is selected so that the number of logic gates in all macro-gates is approximately the same. Figure 3.6 shows a schematic of the segmentation technique, while figure 3.7 presents the pseudo-code of the algorithm. The set of nets that cross the boundary between each pair of layers is monitored during simulation to determine which macro-gates should be activated. We set the values of gap and lid based on mock simulation performance for a small number of cycles.

### 3.5.2 Macro-gate balancing

Each macro-gate is designed to be simulated in a single CUDA multiprocessor. Because our lowest-level primitives are basic logic gates, we designed our CUDA simulation program so that the execution threads simulate all the gates in the same level, then move on to the next level, and so on, until an entire macro-gate has been simulated. Thus the gap is directly proportional to layer simulation performance. However, the segmentation procedure tends

```
1: levelized_netlist = ALAP_schedule(netlist);
2: layers = gap_partition(levelized_netlist);
3: for each layer in layers do
4:     macro − gates = lid_partition(layer)
5:     macro − gates_pool = append(macro − gates);
6:     compute_monitored_nets(layer);
7: end for
```

**Figure 3.7 Macro-gate segmentation algorithm.** The levelized netlist is partitioned into layers: several macro-gates are carved from each layer and appended to the macro-gates pool to be simulated. The nets to be monitored are also tagged at this stage.

to generate macro-gates with a large base (many gates) and a narrow tip. Correspondingly, we have many active threads in the lower levels, and just a few in the top levels.

To maximize concurrency throughout the simulation, we optimize each macro-gate individually with a *balancing* step, as outlined in the schematic of Figure 3.8. This is the last step of the compilation phase: it exploits the slack available in the levelization within each macro-gate and restructures macro-gates to have approximately the same number of logic gates in each level. As a result, a smaller number of threads will be required to simulate the base of the macro-gate. Note that it is always possible to "shrink" the size of the base, at the price of an increased gap.

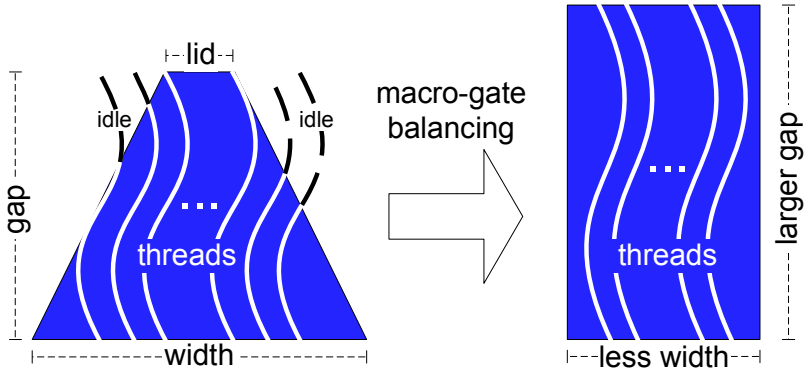



**Figure 3.8 Macro-gate balancing.** The balancing algorithm exploits the levelization slack within a macro-gate to restructure it so that fewer execution threads are required to simulate the lower levels, and idle threads are minimized at the top levels.

### 3.5.3 Simulation phase

As mentioned earlier in this section, simulation is carried out directly on the GPU co-processor. Each multiprocessor is responsible for the simulation of one or more macro-

gates. Each macro-gate corresponds to one thread block. In determining the number of macro-gates that should be simulated concurrently on a multiprocessor, the number of concurrent thread blocks allowed in a multiprocessor (3), was the limiting factor. A single allocation would enable larger macro-gates, however, mapping several smaller ones concurrently allows us to hide the memory latency in retrieving structural netlist data from device memory. We found experimentally that the latter solution provides better performance.

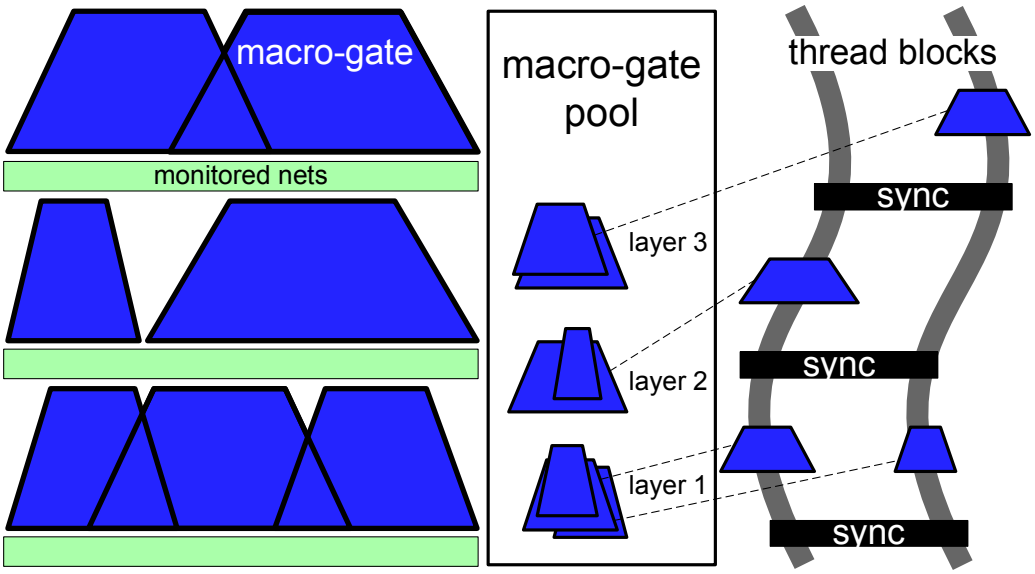



**Figure 3.9 The event-driven simulation** operates by layer. Within each layer, it simulates activated macro-gates and then analyzes the monitored nets to tag additional macro-gates for activation. Activated macro-gates are transferred by the CUDA scheduler to an available multiprocessor for simulation.

The overall simulation alternates executing all active macro-gates in a layer, with analyzing the corresponding monitored nets to determine which macro-gates should be activated for the next layer. The CUDA scheduler is responsible for assigning activated macro-gates to individual multiprocessors. Figure 3.9 illustrates the layered structure of macro-gates and monitored nets. It also shows how activated macro-gates are transferred from the pool to a multiprocessor for execution. Within a macro-gate simulation, multiple concurrent threads simulate all the gates in same level, then synchronize, and finally advance to the next level, until completion.

Data placement is organized as follows: primary inputs, outputs, register values and monitored nets are mapped to device memory, since they must be shared among several macro-gates (multiprocessors). Truth tables for the gates in the technology library are mapped to shared memory because of their frequent access. In addition, intermediate net values generated within a macro-gate are also placed in shared memory. Finally, the netlist structure is stored in device memory and accessed during each macro-gate simulation.

| Design | Testbench | # Gates | # Flops |
|---|---|---|---|
| Alpha no pipeline | recursive Fibonacci program | 17546 | 2795 |
| Alpha pipeline | recursive Fibonacci program | 18222 | 2804 |
| LDPC encoder | random stimulus | 62515 | 0 |
| JPEG decompressor | 1920x1080 image | 93278 | 20741 |
| 3x3 NoC routers | random legal traffic | 64432 | 13698 |
| 4x4 NoC routers | random legal traffic | 144098 | 23875 |
| OpenSPARC core | OpenSPARC regression suite | 262201 | 62001 |
| OpenSPARC-2 cores | OpenSPARC regression suite | 610670 | 124002 |
| OpenSPARC-4 cores | OpenSPARC regression suite | 1221340 | 248004 |

**Table 3.1 Testbench designs for evaluation of the simulator.**

## 3.6 GCS experimental results

We evaluated the performance of our simulator on a broad set of designs ranging from purely combinational circuits such as an LDPC encoder, to a multicore SPARC design containing over 1 million logic gates. Designs were obtained from OpenCores [75] and from the Sun OpenSPARC project [88]; the Alpha processors and NoC designs were developed in advanced digital design courses by student teams at the University of Michigan.

We report in Table 4.1 the key aspects of these designs: number of gates, flip-flops and type of stimulus that was used during simulation. The first two designs are processors implementing the Alpha instruction set, the first can execute one instruction at a time, while the second has a 5-stage pipelined architecture. Both were simulated executing a binary program that computed Fibonacci series recursively. The LDPC encoder outputs an encoded version of its input; for this design we developed a random stimulus generator that run directly on the GPU platform. The JPEG decompressor would decode an input image. The NoC designs consist of a network of 5-channel routers connected in a torus network and simulated with a random stimulus generator sending legal packets through the network. Finally, the OpenSPARC designs use processors from the OpenSPARC T1 multicore chip (excluding caches) and run a conglomeration of assembly regressions provided with Sun's open source distribution. We built several versions of this processor: single-core, two cores, and four cores and we simulated local cache activity by using playback of pre-recorded signal traces from processor-crossbar and processor-cache interactions.

### 3.6.1 Performance of the oblivious simulator

The performance of the oblivious GCS simulator is measured for all testbench designs with the exception of the multi-core versions of the OpenSPARC, due to circuit size limitation inherent in the design of this simulator. The results are discussed in Table 3.2.

| design | cycles | Seq Sim(s) | GCS time(s) | Speed up |
|---|---|---|---|---|
| Alpha no pipeline | 12,889,495 | 40,427 | 9,942 | 4.07x |
| Alpha pipeline | 13,423,608 | 67,560 | 10,688 | 6.32x |
| LDPC encoder | 100,000 | 12,014 | 193 | 62.25x |
| | 1,000,000 | 120,257 | 1,993 | 60.34x |
| | 10,000,000 | >48h | 19,859 | |
| JPEG decompressor | 2,983,674 | 14,740 | 929 | 15.87x |
| 3x3 NoC router | 111,823 | 386 | 50 | 7.72x |
| | 1,225,245 | 2,819 | 324 | 8.7x |
| | 1,967,155 | 4,258 | 504 | 8.45x |
| 4x4 NoC routers | 120,791 | 561 | 82 | 6.84x |
| | 1,298,438 | 3,263 | 424 | 7.7x |
| | 2,018,450 | 5,061 | 659 | 7.68x |
| | 10,000,001 | 34,503 | 4,656 | 7.41x |
| OpenSPARC core - v9allinst.s | 119,017 | 3,221 | 756 | 4.26x |
| - lsu_mbar.s | 137,497 | 3,726 | 880 | 4.23x |
| - lsu_stbar.s | 101,720 | 2,762 | 640 | 4.32x |

**Table 3.2 Oblivious GCS performance.** Comparison of GCS simulation performance against a state-of-the-art event-driven simulator. GCS outperforms the sequential simulator by 14.4x on average.

### 3.6.2 Performance of the event-driven simulator

Finally, we evaluated the performance of our prototype event-driven GCS simulator against that of a commercial, event-driven sequential simulator. Our graphics coprocessor was a CUDA-enabled 8800GT GPU with 14 multiprocessors and 512MB of device memory, operating at 600 MHz for the cores and 900MHz for the memory. The current implementation has 83% occupancy and achieves a bandwidth of 20.4 GB/s. The commercial simulator was run on a 2.4 GHz Intel Core 2 Quad running RH-EL5, enabling 4 parallel simulation threads. For each design, Table 3.3 reports the number of cycles simulated, the runtimes in seconds for both the GPU-based simulator and the commercial simulator (compilation times are excluded), and the relative speedup. Note that our prototype simulator outperforms the commercial simulator by 4 to 44 times. Despite the LDPC encoder having a

very high activation rate, we report the best speedup for this design. As mentioned before, most gates in this design are switching in each cycle: this affects our activation rates, but hampers the sequential simulator performance. Thus, the speedup obtained is due to sheer parallelism of our architecture.

| design | sim cycles | seq sim(s) | GPU sim(s) | speed up |
|---|---|---|---|---|
| Alpha no pipeline | 12,889,495 | 31,678 | 2,567 | **12.15x** |
| Alpha pipeline | 13,423,608 | 54,789 | 7,781 | **7.04x** |
| LDPC encoder | 1,000,000 | 115,671 | 2,578 | **44.87x** |
| | 10,000,000 | >48h | 25,973 | **43.49x** |
| JPEG decompressor | 2,983,674 | 12,146 | 599 | **20.28x** |
| 3x3 NoC routers | 1,967,155 | 3,532 | 397 | **8.90x** |
| 4x4 NoC routers | 10,000,001 | 28,867 | 3,935 | **7.34x** |
| sparc core x1 | 1,074,702 | 27,894 | 6,077 | **4.59x** |
| sparc core x2 | 1,074,702 | 40,378 | 8,229 | **4.91x** |
| sparc core x4 | 1,074,702 | 61,678 | 10,983 | **5.62x** |

**Table 3.3 Event-driven GCS performance.** Performance comparison between event-driven GCS simulator and a commercial event-driven simulator. Our prototype simulator outperforms the commercial simulator by 13 times on average.

## 3.7 Towards high-performance behavioral simulation

One of the most common languages for modeling many digital designs, and particularly embedded systems, is SystemC [74]. SystemC extends C/C++ with libraries to describe HW constructs. It is widely deployed in early-stage analyses and design-space explorations. Unfortunately, simulation performance of SystemC is fairly slow, typically 10x slower than other RTL languages simulations [36]. To make things worse, the most common SystemC simulation kernel (OSCI) uses application-level threading (co-operative threads), thus it is intrinsically sequential because the operating system cannot dispatch co-operative threads to different processing elements. When simulating transaction-level models (TLMs) these limitations do not have a major impact because the scheduler intervenes rarely and does not introduce substantial overhead. In contrast, RTL simulation requires frequent scheduler operations, leading to heavy performance impact.

Simulation solutions for SystemC use an event-based architecture, where a centralized scheduler controls the execution of processes based on events (synchronizations, time notifications or signal value changes). Processes are blocks of activities connected to a same trigger event. Figure 3.10 depicts the execution flow of a typical SystemC simulation ker-

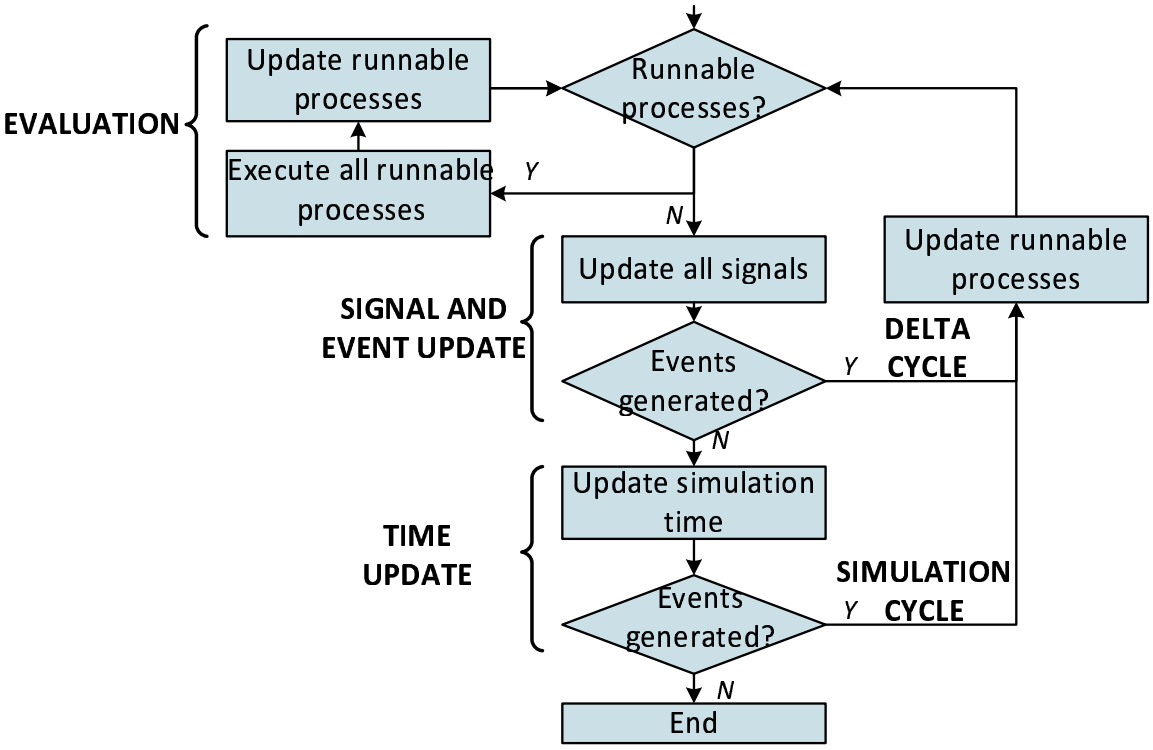



**Figure 3.10 Traditional SystemC simulator scheduler.** The scheduler is the central component of the simulator and it coordinates all activities, including all SystemC processes execution. Its inherently sequential structure makes parallelization of the simulator unattainable.

nel. The flow is iterated until no event is left to be processed, indicating the end of the simulation. A *simulation cycle* completes at the end of each iteration through the complete flow. Within each cycle, there is first an *evaluation phase* during which all runnable processes are executed. Signals are updated at the end of execution of each process. If a signal value change occurs, all processes sensitive to that signal change are added to the runnable queue (this is called *signal and event update phase*).

Finally, during the *time update phase*, the time of the next simulation cycle is determined by setting it to the earliest of (i) the time at which simulation ends, (ii) the next time at which an event occurs, or (iii) the next time at which a process is scheduled to resume. If simulation time is not increased, the next simulation cycle will be a delta cycle. When no new event is fired, simulation ends. The order of process execution within a delta cycle does not affect the simulation's output since the simulator presents the same system's status to all those processes.

The scheduler in a SystemC simulator coordinates the activation of all processes and manages both delta and simulation cycles. Because of this centralized approach, traditional SystemC simulators cannot take advantage of the concurrency of modern CMPs. Hence to

adapt SystemC simulation to a massively parallel platform such as GP-GPUs, we need a very different approach than the one followed by traditional SystemC simulators. The next few Sections (3.8 to 3.9) detail a simulation solution for SystemC RTL targeting GP-GPUs, called SAGA (SystemC Acceleration on GPU Architectures).

## 3.8 Mapping SystemC to GP-GPU

Exposing parallelism in a SystemC simulator is non trivial, since the simulation is neither embarrassingly parallel, nor homogeneous. However, some parallelism can be extracted when treating the active processes in a same delta cycle as concurrent tasks. SAGA exploits this aspect in three steps:

1. construction of the *dependency graph*. We build a static schedule for the processes of the SystemC model under simulation, based upon the signals read and written by each process. The schedule is designed so to lead to equivalent results as the dynamic schedule of the traditional simulator (Section 3.8.1);
2. partitioning of the static schedule into *parallel dataflows*. Dataflows will be executed concurrently in different warps on the CUDA architecture (Section 3.8.2). This step is based on a novel dataflow partitioning mechanism applied to the schedule we generated in the previous step.
3. levelization of processes within each dataflow based on a *sequential order*. The resulting process blocks will be executed by concurrent thread-blocks in the GP-GPU (Section 3.8.3).

The three steps are illustrated in Figure 3.11 and detailed next.

### 3.8.1 Construction of process dependency graph

In *SAGA* we pursue a novel approach to SystemC simulation that allows us to simulate most SystemC designs. The models that cannot use our approach are those that contain circular dependency loops; however, those cannot be synthesized either, so they do not arise in practical designs. In our construction, we arrange the processes in producer-consumer order based on the I/O direction of their connecting signals. To this end, we build a *process-graph PG = (V;E)* where each process is represented by a vertex $V$; a directed edge $E$ from $V_1$ to $V_2$ represents a process dependency due to a signal generated by $V_1$ and consumed by $V_2$. We do not represent synchronous statements in the process-graph, since they create a dependency between present-state values and next-state values through time, which is not represented in our graph. *PG* is a directed acyclic graph (DAG) by construction, and thus we can apply a topological sort to it. Processes dependent only on delta events at

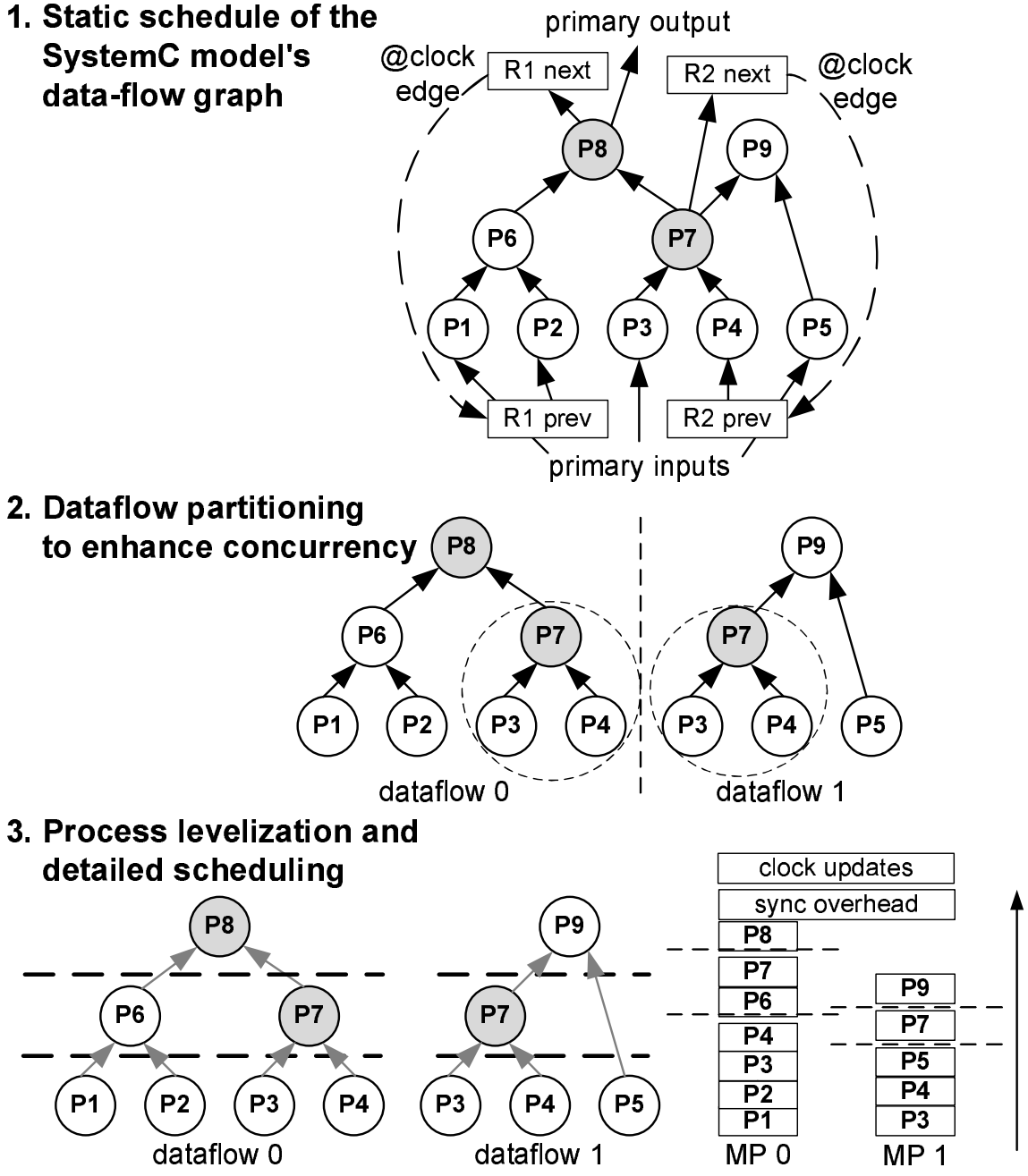



**Figure 3.11** ***SAGA*** **steps in generating a high performance GPU-based SystemC simulation**. *SAGA* proceeds in three steps: first it construct a static schedule for the SystemC processes, than it partitions each dataflow to enhance concurrency, and finally it performs detailed scheduling and mapping.

their primary inputs and synchronous variables occupy the lowest level; the other levels are established by the edge connections.

Figure 3.11.1 shows an example of a process graph built for a typical SystemC module. Nodes in grey represent synchronous processes (*e.g.*, *P8*), while white corresponds to asynchronous processes (*e.g.*, *P6*). Signals R1 and R2 are written by synchronous statements, thus they have a current value (*R1 prev* and *R2 prev* respectively) and a future value (*R1 next* and *R2 next*). Their current value will be updated once the dataflow execution has completed (as suggested by the dashed arrows). Steady-state values at the primary output signals and next state values for the synchronous signals can be obtained by executing the processes level-by-level. Because of how delta cycles operate in a traditional simulator, a PG-based simulation following the schedule we set for the process graph is guaranteed to provide the same results as the traditional simulator at stable state.

Moreover, our construction leveraging static scheduling presents an intrinsic advantage for parallel platforms, since a central event queue structure is no longer needed. Note that we can still benefit from the advantages of an event-driven simulation: all we need to do is check for value-change events at the input of each process within the dataflow. If we only execute a process conditionally to a change at its inputs, then we are basically using an event-based approach and taking advantage of its benefits. This optimization brings upon a 10% performance improvement on average over our baseline solution.

### 3.8.2 Partitioning into concurrent dataflows

There are several ways of partitioning the process graph obtained in the previous section: we select one based on the constraints of our target GPU platform. A straightforward approach would map different processes to distinct threads, one thread per process. We can then execute all processes in a same schedule level concurrently. However, this could lead to severe thread execution divergence if the processes do not share the same source code. Thus, to leverage as much parallelism as possible we devise a novel scheme in which the static schedule of the process graph is partitioned into multiple independent dataflows. These are then mapped to distinct multiprocessors for concurrent execution since different multiprocessors have distinct fetch units. The dataflows we create in this step are segments of the scheduled process graph that can be executed independently. When necessary we may replicate some portions of the process graph to attain independence among dataflows.

The partitioning algorithm is outlined Figure 3.12. First, we select processes in the static schedule that do not activate any other process asynchronously, that is, they are root processes in the PG graph (line 4) (*e.g.*, *P*8 and *P*9 in Figure 3.11.1). For each of these

nodes, we select their fan-in cone in the PG (line 5–12), as illustrated in the second step in Figure 3.11. Processes that are common to multiple cones are replicated in each cone (*e.g.*, the processes in the dashed circles in the Figure) in order to make the cones independent of each other and to enable concurrent execution.

```
 1: list queue;
 2: for each node n ∈ V do
 3:    list current_dataflow;
 4:    if n has no exiting edges then
 5:       queue.add(n);
 6:       while queue is not empty do
 7:          Node current_node = queue.pop();
 8:          current_dataflow.add(current_node);
 9:          for all incoming edges edge of current_node do
10:             queue.add(edge.getSource());
11:          end for
12:       end while
13:    end if
14: end for
15: dataflow_list.add(current_dataflow);
```

**Figure 3.12 Dataflow partitioning algorithm**.

Even though we need to replicate some portions of the process dependency graph, thus increasing the amount of simulation required, replication ultimately eliminates the need of communicating values among dataflows, thus leading to an important reduction in communication cost through device memory.

### 3.8.3 Parallel execution in CUDA

The cones built in the previous step are process dependency trees, that must be executed level-by-level to respect the internal dependency constraints. Thus, for each dataflow obtained in the previous step, we now generate a total serial order of processes that satisfies the level-to-level dependencies.

First of all, we levelize the cones by following the algorithm outlined in Figure 3.13. In this process, if the current node has no incoming edges (and thus it is not activated by any other process in the dataflow), then it belongs to the lowest scheduling level (lines 3–4). Otherwise, the node is scheduled at a level higher than that of all its fan-in processes (line 6-11). This step strengthens the dependency relation between processes (*e.g.*, in the example in step 3 of Figure 3.11, not only *P3* and *P4* execute before *P7*, but also *P5* does).

```
1: for each dataflow dataflow in dataflow_list do
2:    for each node n in dataflow do
3:       if n has no incoming edges then
4:          n.setLevel(0);
5:       else
6:          n.setLevel(-1);
7:       end if
8:    end for
9:    while at least one node has not been assigned a non-negative level do
10:      for each node n in dataflow do
11:         if for each incoming edge edge, the source node edge.getSource() has a non-negative
            level then
12:            for each incoming edge edge of n do
13:               if n.getLevel() < edge.getSource().getLevel() then
14:                  n.setLevel(edge.getSource().getLevel() + 1);
15:               end if
16:            end for
17:         end if
18:      end for
19:   end while
20: end for
```

**Figure 3.13 Dataflow levelization algorithm**.

Then, processes in each dataflow are serialized by starting from the lower levels up to the root processes (processes at the same level can be executed in any sequential order). It is advantageous to create such sequential order for each dataflow, since it eliminates the need of frequent synchronization after each level. An example timeline obtained from this process is shown on the right hand side of Figure 3.11.3.

At this point *SAGA* generates the CUDA code corresponding to the generated process schedule. We use two kernels: a simulation kernel manages dataflow execution, and it is constructed by listing all the dataflows and predicating each by a thread-block ID condition, so that only a specific thread-block is responsible for executing a certain dataflow. The body of each individual process is replaced by equivalent CUDA code, which might require translation of SystemC data-types into native data-types, as reported in Section 3.9.1. The simulation kernel alternates execution with a value-update kernel, responsible for transferring the next-state values into the corresponding present-state values and performing testbench actions. A simulation cycle is completed by one execution of the simulation kernel followed by one execution of the update kernel.

Since device memory accesses are particularly slow, as indicated in Section 3.2, a

key improvement is provided by allocating as little data as possible in global memory. To achieve this, only variables written by synchronous processes are allocated in global memory, since their value must be persistent among different kernel executions. All other variables can be declared as local variables, and will consequently be mapped to registers with much faster access latency.

## 3.9 SAGA experimental evaluation

In this section we evaluate the performance of *SAGA*, provide insights on its intermediate data structure and compare it against other state-of-the art solutions in this space. First we discuss our experimental setup; then compare *SAGA*'s performance against that of a sequential simulator and finally a comparison with other available concurrent solutions is also provided.

### 3.9.1 Experimental setup

*SAGA* considers as input a SystemC design, it transforms it as discussed in Section 3.8, producing all the CUDA code necessary to run the corresponding simulation on a GPU, as output. The code can then be off-loaded to a GPU platform and executed. All experiments discussed below were evaluated on a NVIDIA GTX480 GPU and a Intel quad core i7 operating at 2.8Ghz and running Linux RedHat 5.7. In addition, we leveraged the HIFSuite framework [37] to parse the SystemC code and generate an intermediate data structure that is used by *SAGA* for its internal transformations.

The first task in *SAGA* consists of considering a SystemC description and translating it into the HIFSuite's internal format (HIF) by using the HIFSuite *sc2hif* tool. The code generated at this point is a tree-structured XML-like representation of the original code, where semantic objects are represented with TAGS.

*SAGA* then applies a number of pre-processing steps to the HIF description. First it extracts all the processes and builds an initial dependency graph, according to signal dependencies among processes. It then applies the 3-step transformation described in Section 3.8.

At this stage SystemC data types are substituted with native C/C++ data types and all corresponding data structures are built. This transformation is necessary because the CUDA language does not support SystemC data types. Finally, *SAGA* generates the code for the kernel functions, and outputs the generated HIF description representing the de-

tailed scheduled dataflows obtained with our algorithm. As a last step, the resulting HIF code is converted into C code by means of the HIFSuite *hif2c* tool. This representation is ready to be compiled for the target CUDA architecture.

Table 3.4 presents our testbench designs. The designs are part of a complex embedded platform that was developed in the context of a European project together with silicon vendor industry partners. Specifically:

- ECC is an error correction code device.
- ClockGen, ResGen, Sync and RegCtrl are part of a complex DSPI system. ClockGen is a clock generator, creating multiple clocks for the various components in the system. ResGen transforms and outputs the computed results in the specified format. Sync is a specialized synchronization function among a number of components. RegCtrl is a register controller for a set of registers.
- 8b10b is a module performing encoding and decoding byte-wide data according to the 8b/10b protocol.

We evaluated *SAGA* on the individual testbench designs and on two more complex SoC design assemblies: Half Platform, comprising ECC, ClockGen, ResGen and Sync; and Plaform integrating all the testbench designs previously discussed. For each design, Table 3.4 reports the number of processes in the original SystemC description (*Processes (#)*), the lines of code (*SystemC (loc)*), the number of dataflows extracted (*Dataflows (#)*) and the amount of code replication due to our step 2 (see Section 3.8) measured in number of processes replicated (*Replicated processes (#)*).

| Design | Processes(#) | | SystemC | Dataflows | Replicated |
|---|---|---|---|---|---|
| | Synchr. | Asynchr. | (loc) | (#) | processes (#) |
| ECC | 4 | 7 | 582 | 4 | 4 / 3 |
| ClockGen | 6 | 15 | 741 | 12 | 7 / 3 |
| ResGen | 3 | 6 | 478 | 9 | 0 / 0 |
| Sync | 4 | 22 | 641 | 23 | 0 / 0 |
| RegCtrl | 18 | 32 | 2677 | 43 | 17 / 8 |
| 8b10b | 7 | 30 | 799 | 7 | 9 / 3 |
| Half Platform | 18 | 51 | 2355 | 48 | 11 / 3 |
| Platform | 42 | 112 | 5643 | 98 | 37 / 8 |

**Table 3.4 Characteristics of the testbench designs used for *SAGA*'s evaluation**.

### 3.9.2 Performance

Table 3.5 compares *SAGA*'s performance with that of a SystemC sequential execution as discussed in Section 3.7. For each design, Table 3.5 reports simulation time of the SystemC simulation (Column *SystemC simul. (ms)*) and of the *SAGA*-generated CUDA code (Column *SAGA simul. (ms)*). It then reports their comparative performance in terms of *SAGA*'s speedup over sequential execution (Column *Speedup (x)*). The results show that the *SAGA* simulation is always faster than its corresponding SystemC sequential simulation. However, the speedup is moderate when comparing the small, individual component designs, leading to up to a 3.89 times improvement. Note, however, that even in presence of highly heterogeneous and complex processes *SAGA* achieves a respectable performance improvement. In addition the speedup achieved with the two more complex designs is much higher, ranging from 10 to almost 16x. This result suggests that *SAGA* is a promising solution that can extract even more concurrency from the more complex designs where there are more processes available, leading to a better utilization of the parallel resources available on the GP-GPU.

The speedup achieved by *SAGA* is bounded both by the amount of concurrency that can be extracted from each module, and by the amount of computation they require. When both these factors are high, the generated code greatly outperforms sequential SystemC simulation. A low level of parallelism (ECC and ClockGen) or non-intensive computation (ResGen and Sync) lead to lower speedups, due to a heavier contribution of synchronization not balanced by computation, or because the limited concurrency is not offset by its setup overhead.

| Design | SystemC simul. (ms) | *SAGA* simul. (ms) | Speedup (x) |
|---|---|---|---|
| ECC | 11.99 | 5.05 | 2.37 |
| ClockGen | 18.00 | 7.13 | 2.52 |
| ResGen | 8.97 | 5.22 | 1.71 |
| Sync | 9.98 | 5.73 | 1.74 |
| RegCtrl | 41.97 | 13.05 | 3.21 |
| 8b10b | 15.99 | 4.11 | 3.89 |
| Half Platform | 83.98 | 8.143 | 10.31 |
| Platform | 228.96 | 14.34 | 15.97 |

**Table 3.5 Performance improvement of *SAGA* vs. a sequential simulator**.

### 3.9.3 Architecture comparison

In order to show the effectiveness of the proposed methodology, we compared the performance of *SAGA* against that of two other concurrent solutions for SystemC simulation. For this study we report results on only two designs for sake of brevity. However, these two designs are representative of typical behavior and we found that the other designs lead to similar outcomes.

For this study we considered a concurrent SystemC simulator targeting a standard chip multiprocessor (CMP) and also another GPU-target simulation solution, called SCGPsim [71], outlined in Section 2. We implemented SCGPsim based on their description and we developed the CMP solution using the `pthread` library to map SystemC processes. We report our findings in Table 3.6, where speedups are normalized to the performance of the sequential simulator.

The table indicates that *SAGA* is the fastest solution, providing a speedup of 2 to 4x over the solution of [71], and even more over the multiprocessor design. Also note that the other solutions do not provide a performance improvement over the sequential simulation for ECC. Upon further inspection we found that the CMP solution does not achieve good concurrency because distinct processes are mapped to co-operative threads, as discussed in Section 1. SCGPsim's performance is not high because of the reasons discussed in our related work section: unless processes share the same code, they are scheduled sequentially when mapped on a same multiprocessor, since each multiprocessor uses a single fetch unit. We believe that the authors of [71] experienced much higher speedups because they evaluated their solution on SystemC descriptions where processes had identical code. However, this is a very rare situation for any practical design.

| Implementation | ECC | | ClockGen | |
|---|---|---|---|---|
| | Time (ms) | Speedup | Time (ms) | Speedup |
| SystemC | 11.99 | 1x | 18.00 | 1x |
| Multiprocessor | 94.00 | 0.13x | 20.00 | 0.9x |
| SCGPsim | 20.08 | 0.59x | 14.77 | 1.22x |
| *SAGA* | 5.05 | 2.37x | 7.13 | 2.52x |

**Table 3.6 Performance comparison of *SAGA* vs. other concurrent solutions**: Multiprocessor is a concurrent simulator using pthreads on a CMP; SCGPsim is a parallel simulator targeting GPUs.

## 3.10 Related work

Research on logic simulators bloomed in the 1980s, when the concepts of circuit netlist compilation, oblivious and event-driven simulation were first explored [21, 12, 60, 11].

In particular, [11] provides a comparative analysis of early attempts to parallelize event-driven simulation by dividing the processing of individual events across multiple machines with fine granularity. This fine granularity would generate a high communication overhead and, depending on the solution, the issue of deadlock avoidance could require specialized event handling. Parallel logic simulation algorithms were also proposed for distributed systems [66, 64] and multiprocessors [55]. In these solutions, individual execution threads would operate on distinct netlist clusters and communicate in an event-driven fashion, with a thread being activated if switching activity was observed at the inputs of its netlist cluster. Both conservative [25, 70, 42] and speculative techniques, such as time warp [16, 14], were proposed to handle synchronization in these discrete event algorithms. Today, several commercial simulators building on these concepts are available: they execute on a single CPU and adopt aggressive compiled-code optimization techniques to boost their performance.

In addition, specialized hardware solutions (*emulation systems*) have also been explored to boost simulation performance. These systems typically consist of several identical hardware units connected together, with units optimized for the simulation of small logic blocks. To emulate a circuit netlist, a "compiler" partitions the netlist into blocks and then loads each block into separate units [34, 10, 57]. Modern emulators can deliver 3-4 orders of magnitude speedup and they can handle very large designs. However, their cost is prohibitively large and the process of successfully mapping a netlist to an emulator can take up to few months.

Logic simulation has been attempted on vector processors in the past[52, 81]. Most recently, a few research solutions have been proposed to run simulations on GPUs: a first attempt by Perinkulam [78] did not provide performance benefits due to lack of general purpose programming primitives for their platform and the high communication overhead generated by their solution. Another recent solution in this space [44] introduces parallel fault simulation on a CUDA GPU target. It derives its parallelism by simulating distinct fault patterns on distinct processing units, with no partitioning within individual simulations or the design. In contrast, we target fast simulation of complex designs, thus we must explore circuit partitioning and optimizations techniques in order to leverage the parallelism of the target platform. Moreover, we optimize the performance of individual simulation runs, in contrast with [44], which optimizes over all faults simulations.

Several works in the literature proposes to take advantage of the inherent parallelism of SystemC processes to speedup simulation [28, 39, 98, 83]. Most of them exploit the fact that the order of execution of processes activated within the same simulation cycle does not affect simulation's results. Thanks to this characteristic of SystemC scheduling, processes that are activated within the same delta cycle can be executed in parallel, either

by using multiple threads or by designing a distributed scheduler. For instance, in [39] SystemC processes are executed as threads on multiple CPUs. Each CPU is assigned a thread and an execution environment and then a set of runnable processes is assigned to each thread. Unfortunately, simulation relies on a simulation platform (ArchSim) that introduces a lot of overhead, thus making this approach ineffective. In contrast, the approach of [28] uses a distributed scheduler. Each processing node includes a copy of the scheduler and it simulates a subset of the application modules. All scheduler's copies must synchronize after each delta cycle to update the value of shared signals and of simulation time, thus this approach generates many synchronization events among processes running on separate processors.

A largely different approach is proposed in [83], which transforms modules' structure. The methodology analyzes SystemC modules and it identifies blocks within processes that can be executed within one phase of the SystemC simulator. Then these blocks are scheduled according to their data and control dependencies. The result operates equivalently to a concurrent scheduler, with the difference that this was achieved via static code analysis. All these solutions rely on code's modifications or introduce heavy overhead because they rely on the existing simulator architecture [98, 28].

A different approach is proposed by the authors of [71], who also target the massive parallelism offered by today's GP-GPUs. In their solution, independent SystemC processes are mapped into parallel threads that synchronize at each iteration of a delta cycle (Figure 3.10) through a barrier synchronization to maintain the correct producer-consumer relation among threads. Since typical SystemC processes contain few word-level and arithmetic operations, this can lead to more time spent on synchronization than execution.

## 3.11 Summary

This chapter demonstrates that the performance of software-based simulation can be improved by using an alternative execution substrate in the form of GP-GPUs. Simulation of digital designs at different abstraction levels is essentially a parallel computation problem which can be accelerated via parallel processing. The massive execution parallelism of GP-GPUs turns out to be a good fit for this problem. The potential of leveraging GP-GPUs for accelerating simulation is explored at two abstraction levels in particular, gate-level logic simulation and a subset of SystemC behavioral descriptions.

Towards accelerating structural logic simulation, two novel gate-level simulator architectures were developed that leverage the high degree of parallelism of GP-GPUs. By

extracting parallelism in the simulation of gate-level netlists, we are able to realize a 13 times speedup over traditional sequential simulators, on average. The oblivious simulator maps complex netlists to the GPU by employing a novel clustering and balancing algorithm. The algorithm cleverly orchestrates the use of GPU resources to convert their high computing power into simulation performance. While the event-driven simulator carves out macro-gates from the structural netlist of a design and schedules them for simulation on the multiprocessors of the NVIDIA CUDA architecture, only if they are activated by switching events at their inputs.

This chapter also demonstrated the viability of GP-GPUs as an accelerator for software-based simulation at a much higher level of design description, namely SystemC. This problem is more challenging as the computation pattern is even more irregular compared to gate-level simulation. To tackle this challenge, we proposed novel static data-flow partitioning algorithms to extract the parallelism present in the problem to map it to the parallelism available in GP-GPUs. This scheme allows us to forgo frequent synchronizations and deliver better simulation performance. We achieved up to an order-of-magnitude speedup over conventional SystemC simulators.

Overall, The experimental results show that the discussed software-based simulators on GP-GPU execution substrate are capable of delivering a remarkable performance speedup on large, industrial-scale designs over existing software-based simulators, thus bringing about new validation frontiers for the digital design industry. However, referring to the simulation spectrum as explained in Chapter 2, software-based simulation acceleration solutions as described in this chapter are still slower than dedicated hardware-accelerated platforms. Unfortunately, the hardware-accelerated platforms do not provide the same degree of checking and debugging capability as possible with software-simulators, hence the performance advantage is not fully harnessed for validation. Hence, to achieve validation at the highest performance it is imperative to bring in such capability to those platforms. This will be the guiding motivation for next few chapters: to achieve checking and debugging capability on hardware-accelerated simulation platforms.

# Chapter 4

# Providing Observability for Hardware-accelerated Simulation

All platforms for simulation-based validation beyond software-based simulation are plagued by the fundamental limitation of lack of observability. As discussed in Chapter 2, we can only trace a subset of signals in acceleration and emulation platforms for simulation performance reasons; while in silicon such tracing capabilities incur chip area overhead. Since simulation performance deteriorates with the amount of recorded data, it is imperative that only a small number of signals are selected for tracing. However, in order to debug a design we often need to know the value of many internal signals. An approach towards solving this problem involves recording the values of a small number of signals and reconstructing the values of several non-observed signals from this information, which in turn may facilitate debugging. This approach necessitates heuristics and algorithms to find a set of signals that have the potential of reconstructing the maximum number of non-observed signals. This chapter of the dissertation presents a simulation-based method to evaluate such reconstruction potential of subsets of signals, leading to selection of a subset which is most beneficial from this perspective.

## 4.1 Towards obtaining observability beyond software-based simulation

As discussed in Chapter 2, acceleration and emulation platforms incur performance cost for observation or recording of internal design signals. Hence, only a small group of signals are usually selected for observation. This problem is even more acute in the post-silicon validation phase. The capabilities of physical probing tools [72] are very limited, and it is infeasible to observe each and every signal in fabricated silicon. So far, reusing *design for test* (DFT) circuit structures, such as internal scan chains, for silicon debug has been widely adopted in the industry [92]. Though scan chains can capture all or a subset

of internal state elements, and thus increase signal observability for silicon debug, it may take several thousand clock cycles to dump out one observed state snapshot and, in most cases, the circuit's execution must be suspended until the completion of this process. The **common fundamental challenge lies in the very limited visibility of internal design signals**.

To facilitate debugging under acceleration, emulation or the silicon itself, ***design for debug* (DFD) structures such as embedded logic analyzers (ELAs)**, have been proposed [3] and have found widespread use in the industry [6, 95, 7]. An ELA consists of a mix of trigger units and sampling units. Programmable trigger units are used to specify an event for triggering the logging of internal signal values. Sampling units are used to log the values of a small set of signals (trace signals) over a specified number of clock cycles into trace buffers. The number of signals traced is known as the *width* of the trace buffer, while the length of the tracing interval is called *depth*. Trace buffers are implemented with on-chip embedded memories [95] and data acquisition can be performed during normal chip operation by setting up the relevant trigger event. Subsequently, the sampled data is transferred off-platform via low bandwidth interfaces for post-processing analysis for debug. Note that **DFD structures must maintain a low logic/area overhead profile**, since they do not provide added benefits to the design. As a result, **only a very small number of signals can be traced** in comparison to those available in the design.

For ELAs to be effective, designers must carefully select for tracing those signals that yield the most debug information. **Through a judicious choice of trace signals, one can even reconstruct data for state elements that are not traced**. As an example, for microprocessor designs, it is common practice to trace pipeline control signals so that the values of other data registers can be inferred during post-analysis. This approach cannot be used for a general circuit, however, because it leverages architectural knowledge of the design. Indeed, the need for generalized solutions in this domain is growing.

Even though the additional inferred information does not guarantee identification of design errors, it still increases internal signal visibility and has the potential of providing valuable debugging information. Because functional bugs tend to occur in unexpected regions and configurations, it is not always possible to predict the most important signals to trace. Ideally we would like a mechanism which allows **reconstructing almost all internal signals from the tracing of just a handful of signals, so as to offer comparable quality of observability** in hardware accelerated platforms or the silicon itself, as offered by software-based simulation.

Recent research addressing these challenges [58] has shown that many non-traced signals and state elements can be inferred from a small number of traced state elements by

forward and backward implication, even in arbitrary logic. Ko and Nicolici [58] were first to propose an automated trace signal selection method that attempts to maximize the number of non-traced states restored from a given number of traced state elements. **The restoration process can also be considered as a data compression technique in an inverse way**. The information content of all restored signals is compressed in the traced signals in a lossless fashion. **The quality of the trace signal selection was quantified by the state restoration ratio (SRR), that is, the ratio of the number of state values restored over the state values traced, over a given time interval**. This measure has been adopted by subsequent research to compare the quality of other solutions. Further research [61, 80, 13] has proposed several automated trace signal selection methods based on different heuristics for estimating the state restoration capabilities of a group of signals. These research solutions share a common structure: (i) a metric to estimate the state restoration capability of a set of state elements and (ii) the use of the metric in a greedy selection process to evaluate candidate set of signals and converge to a final selection.

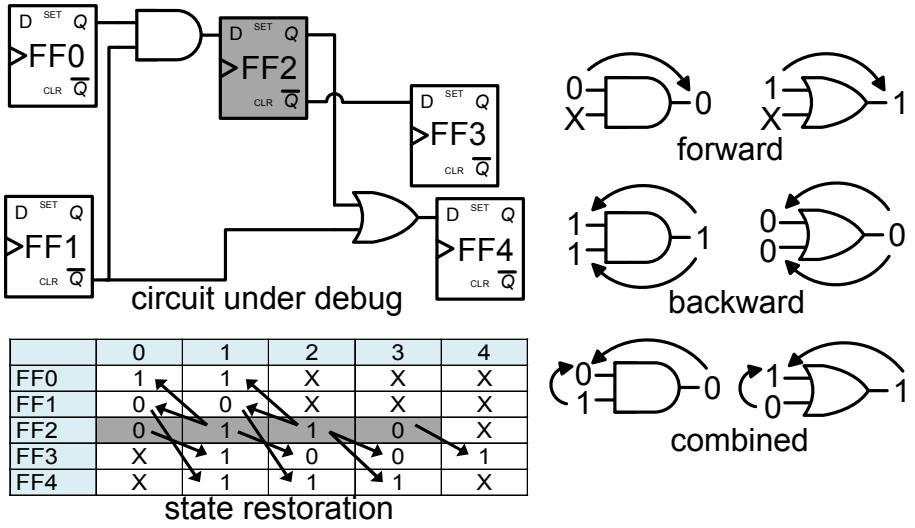
### 4.1.1 Overview of this chapter

In this chapter, first we provide the background of the state restoration process (Section 4.2) and then present the common structure and shortcomings of existing signal selection algorithms with the objective of maximizing restoration (Section 4.3). In Section 4.4, we demonstrate that an **accurate metric for state restoration capability of a set of signals** can be obtained by actually simulating the restoration process on the circuit over a small number of cycles, and measuring the corresponding restoration ratio. Then a novel signal selection method guided by this metric is presented in Section 4.5. This solution **overcomes a key shortcoming of previous greedy approaches to a large degree, namely that of diminishing returns**: when the number of traced signals is increased, additional restored state elements increases sub-linearly. Effectively this solution is able to provide a higher degree of observability into the design, which will greatly facilitate debug. This is demonstrated in the experimental results presented in Section 4.6. Relevant prior work is presented in Section 4.7 and finally the chapter is concluded with Section 4.8.

## 4.2 Background of state restoration

An ideal debugging solution for platforms beyond software-based simulation would allow the same level of observability i.e. every signal value is observable at each cycle, with little

design effort and area overhead. A more realistic goal is to attain partial observability by tracing a small set of signals and use them to find the root cause of the bug. Several previous solutions have suggested automatic signal selection algorithms to determine which state elements allow maximum restoration if traced. An intuitive measure for evaluating restoration quality is the state restoration ratio, defined as $SRR = \frac{N_{traced}+N_{restored}}{N_{traced}}$, where $N_{traced}$ is the number of traced state elements and $N_{restored}$ is the number of restored ones during the time window dictated by the trace buffer's depth. Automated signal selection strives to maximize *SRR*.




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| FF0 | 1 | 1 | X | X | X |
| FF1 | 0 | 0 | X | X | X |
| FF2 | 0 | 1 | 1 | 0 | X |
| FF3 | X | 1 | 0 | 0 | 1 |
| FF4 | X | 1 | 1 | 1 | X |

**Figure 4.1 Example of state restoration process**. The circuit shown at the top left is the circuit under debug, with flip-flop FF2 traced for 4 clock cycles (shown in grey). The table below lists the values of all flip-flops, whether traced, restored or unknown(X). Forward inference and backward justification through the logic gates (shown with forward and backward arrows in the table) allows to restore several flip-flop values that were not traced. The elementary rules of forward inference, backward justification and combined inference are shown for two types of logic gates on the right side of the figure.

The state restoration process relies on the special Boolean property that if a controlling value is known for at least one input of a logic gate, the output can be inferred without the knowledge of other inputs. This property is used for forward inference of signal values in the case of partial knowledge. Similarly, if a non-controlled value is observed on the output of a gate, all input values can be inferred to be the non-controlling value for that type of gate, enabling backward justification. Combined inferences leveraging knowledge of both inputs and output are also possible. Repeated application of these simple operations for all gates of a circuit till no new value can be generated at any signal leads to value reconstruction for state elements beside those traced. This process is used in post-analysis of the data obtained from trace-buffers to restore other non-traced signals. Figure 4.1 illustrates this process with an example inspired by [58]. In this example flip-flop FF2 is traced over

four clock cycles; additional values at other flip-flops can be inferred as shown in the table in the lower part of the figure. In this particular example, the state restoration ratio (SRR) is $SRR = 15/4 = 3.75$ ($N_{traced} = 4, N_{restored} = 11$). Authors of [58] introduce an efficient bit-parallel algorithm to perform this restoration process, which we extensively use in our implementation. It is important to note that the forward inference and backward justification operations are correct only if the logic functions of the gates in the circuit conform to the structural netlist, with no stuck-at-faults or other such electrical faults. Timing errors must also be avoided for correct restoration, a goal that can be attained by reducing the clock frequency during debug operations for silicon. Hence this technique is only effective for investigating functional bugs. The key challenge of this process is how to select which state elements to trace among the thousands of a typical design to achieve the best possible restoration of internal signals and other state elements.

## 4.3 Structure of existing signal selection algorithms

The signal selection algorithms presented in the literature so far [59, 61, 80, 13] focus on delivering maximal restoration ratio and share a common structure. First, a metric is devised to estimate the capacity of state restoration of a given set of signals; second, a greedy selection process guided by the metric converges to a locally-optimal selection. Figure 4.2 summarizes this general structure.

**Input:** *circuit*, width of trace buffer $w$, restoration capacity metric $f_C(...)$
**Output:** selected flip-flop set $T$

```
1:  while |T| < w do
2:     maximum observability maxV = 0;
3:     for each unselected flip-flop s in circuit do
4:        T=T ∪ {s};
5:        observability V=f_c(T);
6:        T=T − {s};
7:        if V > maxV then
8:           selected = s;
9:           maxV = V;
10:       end if
11:    end for
12:    T=T ∪ {selected};
13: end while
```

**Figure 4.2 Pseudo-code for the general structure of greedy automatic signal selection algorithms.**

For the algorithm to be successful the capacity metric should have the following prop-

erties: (i) it should be proportional to the actual average *SRR* that can be obtained with the given set of signals over many runs, (ii) it should be as computationally inexpensive as possible, since several such computations will be needed in the final selection process. The first criterion is especially important for the greedy selection process to be successful, since it guides the successive greedy choices towards the optimal subset. The greedy selection process starts off with the signal which promises the maximum capacity and then enlarges the set one signal at a time by evaluating the restoration capacity of all possible candidate sets with one more signal. In Section 4.4 we will explore how a better capacity metric can be obtained by simulated restoration, while a critical shortcoming of the greedy selection process itself is detailed in next section.

### 4.3.1 The problem of diminishing return with greedy selection

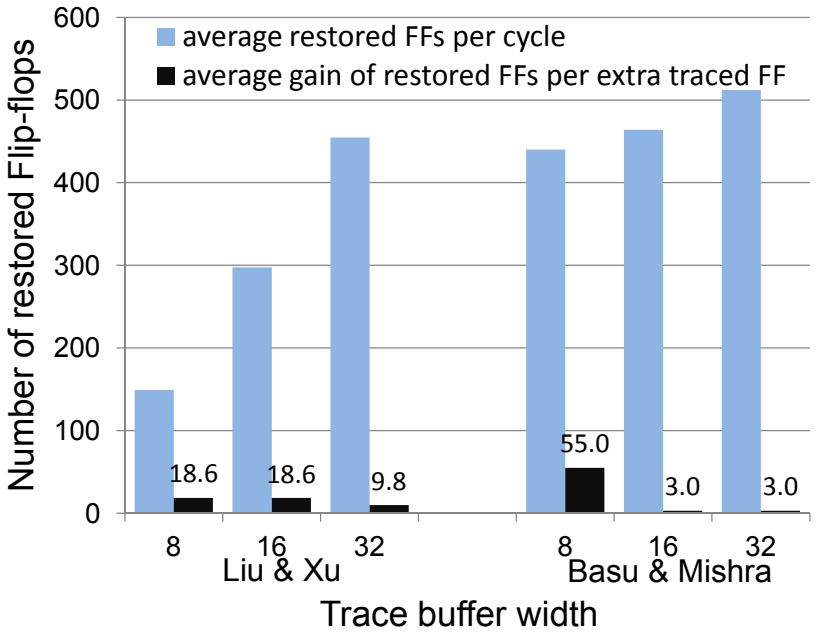



**Figure 4.3 Diminishing return of number of restored flip-flops with increasing trace buffer size** is observed for two previous solutions. The plots are corresponding to circuit s38417.

The greedy selection process adopted in the previous solutions suffer from another critical problem with regards to the quality of the final set of signals chosen. Figure 4.3 plots the average number of restored flip-flops per cycle for 3 different width of the trace buffer (8,16,32) for the ISCAS89 benchmark circuit s38417. Alongside the average number of restored flip-flops gained by addition of each new traced flip-flop is plotted as well. The plots correspond to the data reported by Liu and Xu [61] and by Basu and Mishra [13]. Note that in the result obtained by Liu and Xu, growing the number of observed flip-flops

from 8 to 16 increases the average number of restored flip-flops per cycle, from 149 to 298, which is a good rate ($(298-149)/(16-8) = 18.62$) of gain of information per added new trace signal, shown in the adjacent dark bar. However when the number of traced signals is increased from 16 to 32, the rate of gain is much lower. This effect is more pronounced in the results obtained by Basu and Mishra [13], where a much better initial set of signals is obtained but as the number of trace signals are doubled, the gain in the average number of restored flip-flops is very minute. This behavior results from inaccuracy in the estimation metric and due to the very nature of the greedy selection. The greedy selection algorithm starts off with the flip-flop promising maximum restoration and attempts to grow the set by one flip-flop at a time, and the average number of restored flip-flops plateaus off when a larger number of flip-flops are traced. When choosing $2n$ flip-flops, the choice is already constrained by previously chosen $n$ flip-flops: We have to keep the $n$ chosen flip-flops in the set and find additional flip-flops which when added with the existing set provides maximum restoration possible under this constraint. However the best possible set of $2n$ flip-flops might not have all the $n$ flip-flops, since there might be other $n+1$ or more flip-flops which when taken together are able to restore more missing signals, but would not be able to enter the final selection, since the algorithm only makes greedy choices in the forward direction trying to grow a pre-decided set of $n$ flip-flops. Hence for choosing a larger number of traced signals an alternative approach of making greedy decisions from the backward direction, i.e. starting off with the set of all flip-flops and then constraining the set slowly to the required width, can be more successful. We outline an algorithm to perform this elimination process.

## 4.4 Improving restoration capacity metric

As mentioned earlier, a good restoration capacity metric should possess high degree of correlation with the actual observed SRR obtained with a set of signals. Since, the more accurate the metric, the more likely it is to arrive at the optimal subset of signals at the end of selection process. To evaluate the quality of a restoration capacity metric, we devise the following experiment. For a design we choose 1000 random sets of 8 flip-flops each and measure the average SRR per group, for a trace buffer depth of 4096, obtained with 100 simulation runs (using 10 sets of random seeds and 10 different starting point of tracing i.e. offset from the initial circuit reset state, per seed). It is ensured that the circuit remains in functional mode during the entire tracing process, by asserting appropriate value at reset and other control signals. We can now plot the average SRR versus the estimated

state observability obtained with a restoration capacity estimation metric in a scatter plot to measure the correlation of the metric with actual measured SRR.

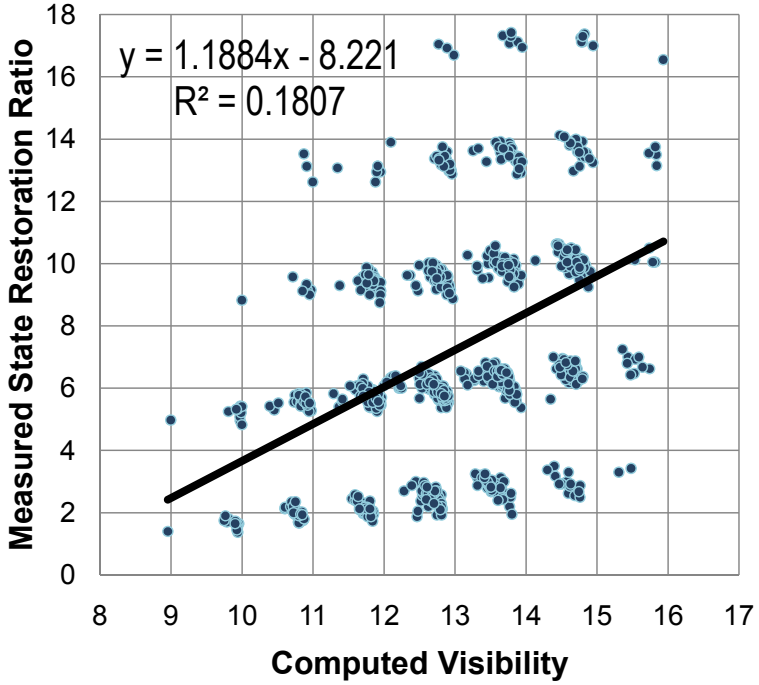



**Figure 4.4 Correlation of restoration capacity metric described by Liu and Xu with measured SRR for circuit s35932.** The metric has poor yet positive correlation with measured SRR. Note that data points in the bottom right corner represents selection of flip-flops that have a high estimated value of state observability but rather poor measured SRR. This behavior can drive the greedy selection algorithm to sub-optimal selections. A linear regression fit of the data is shown in the plot, along with square of the correlation coefficient.

We implemented the restoration capacity metric called observability V, described by Liu and Xu [61]. Figure 4.4 shows the correlation of this metric with observed SRR. As seen in the figure, though this metric has positive correlation with measured SRR, the extent of correlation is poor; as indicated by a low value of the correlation coefficient($R$). Also this metric can over-estimate as well as under-estimate the SRR of certain selections leading to a sub-optimal final selection. The fundamental reason behind this behavior is lossy information compaction in probability based restorability estimates. Consider the two input AND gate in Figure 4.5, where the restoration probability of value 1 at the both inputs are known to be 0.5 and no other knowledge is present. A probability based estimation scheme will infer the restoration probability of value 1 at the output to be $0.5 \times 0.5 = 0.25$. However if the actual restored value in the two signals over 6 successive clock cycles are $1X1X1X$ and $X1X1X1$, both in accordance with the estimated restoration probability, though we can not restore the output for any of the cycles. This flaw is common to all probability based estimates and the inaccuracy results from compaction of information that is spread across several cycles into a single number, and could be avoided if we had a conditional proba-

bility distribution of each signal's restorability given the value of other signals. However such detailed probabilistic treatment is infeasible. This example shows that the restoration probability estimates are not reliable, and often do not correlate well with actual restoration behavior.

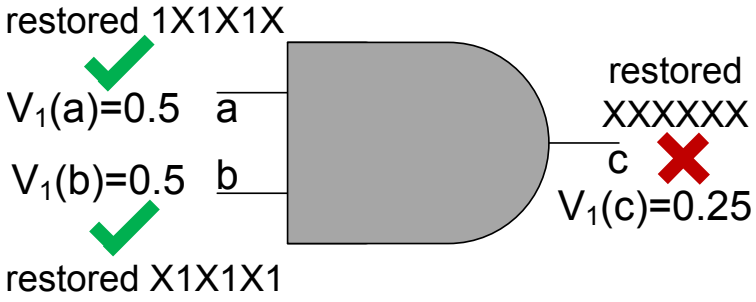



**Figure 4.5 Restoration probability estimates can be misleading**, as seen in this example.

Keeping the ideal characteristics of a restoration capacity metric in mind, we investigated whether a metric of restoration capacity can be constructed out of simulation of restoration itself. The best estimate of SRR for a group of traced signals and trace depth in a circuit can be obtained by performing a large number of simulations with different random seeds (for generating inputs) and starting tracing at several random offsets from the initial reset state, then performing the restoration process for the circuit, finally taking the average of the SRR values from each individual simulation. This is effectively analogous to performing Monte-Carlo simulations for obtaining an estimate of SRR for a group of traced signals. However, even though this estimate would be extremely accurate, each of the individual simulations (also includes the restoration process per simulation) takes up a considerable amount of execution time when performed for typical trace buffer depth ($\sim$4K clock cycles) and also several such simulations will be needed to establish a single estimate. This violates the second criterion of an ideal capacity estimation metric. A selection algorithm will need a large number of such estimates to converge on to the final set of signals, hence if each of the individual estimations are computationally intensive the overall selection process would demand an inordinate amount of time for any realistic circuit size.

A key insight to solve this problem is the fact that the estimate of state restoration capacity does not need to exactly match observed SRR, it only has to be highly correlated with the actual SRR that can be obtained with the same group of traced signals. A common method of reducing effort in simulation based estimation is to perform several short simulations and average their results. In this particular case which amounts to performing the state restoration process but for a smaller length of the trace buffer. This observation lead us to carry out a study about sensitivity of SRR on varying depth of the trace buffer. The results

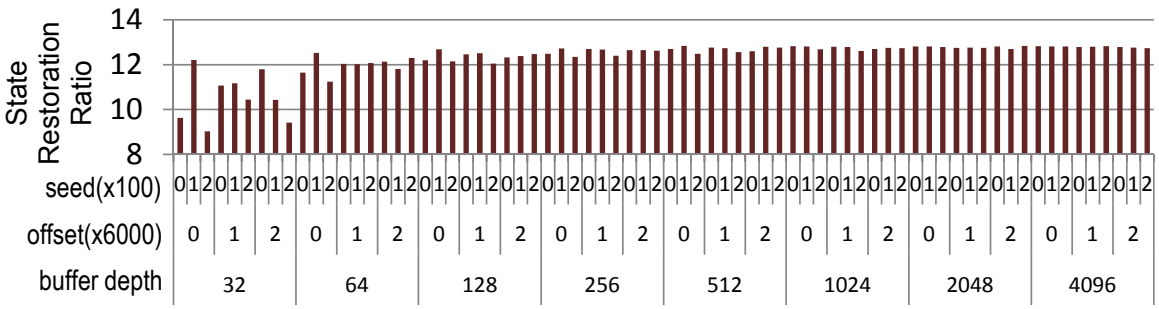



**Figure 4.6 Variation of SRR with trace buffer depth** (3 random offsets per case, 3 random simulation seeds per offset for the s35932 circuit). The value of the observed SRR for a group of signals is fairly insensitive to buffer depth beyond 64.

for a certain selection of 8 flip-flops in s35932 circuit is shown in Figure 4.6. For purpose of legible representation only 9 random samples per trace buffer depth are displayed: 3 different random offsets and 3 random seeds per offset. The main observation from this study is that the value of the SRR obtained from a certain group of traced signals is fairly insensitive to depth of the trace buffer. In fact, there is very little variation beyond the depth of 64 cycles. Similar behavior is observed for all other circuits, as well as when more random samples are obtained. This observation suggests that measured SRR from simulated restoration for small depths (∼64) can serve as an estimation metric of restoration capacity.

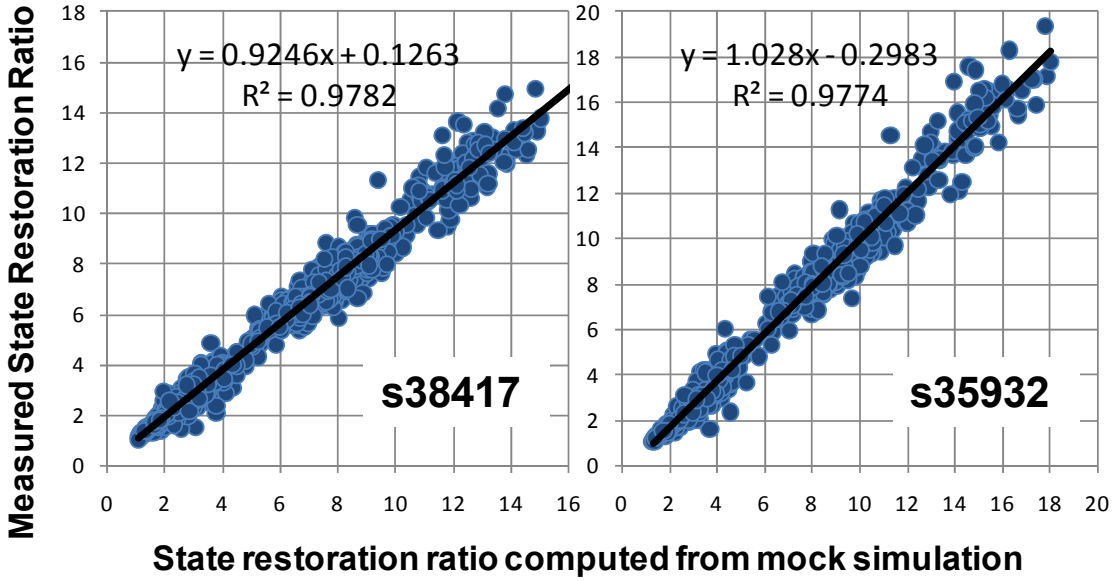



**Figure 4.7 Correlation of observed SRR with our proposed restoration capacity metric namely, SRR obtained from mock simulation with 64 cycle of buffer depth.** Correlation is shown for two circuits: s38417 and s35932. The proposed metric bears strong positive correlation with the observed SRR indicated by the value of the correlation coefficient.

The hypothesis that SRR obtained from mock simulated restoration for small depths has good correlation with the observed SRR is further validated by repeating the earlier correla-

tion study except for plotting the simulation based metric on the X axis in this case for two benchmark circuits. The mock simulation uses depth of only 64 cycles with one random seed and one random offset, a convention used for all estimation purposes described in the rest of the chapter. The resultant scatter plot for circuits s38417 and s35932 is shown in Figure 4.7. The simulation based capacity estimation evidently shows an extremely high degree of linear correlation with the observed SRR. Similar strong correlation was found for other circuits as well. This observation confirms the viability of using SRR obtained from mock simulation of restoration for a small depth as an accurate estimate of restorability of state elements. Note that, a larger depth and averaging over more random seeds and offset values will make the estimate even more accurate and should be deployed if more compute resources are available.

## 4.5 Proposed signal selection algorithm

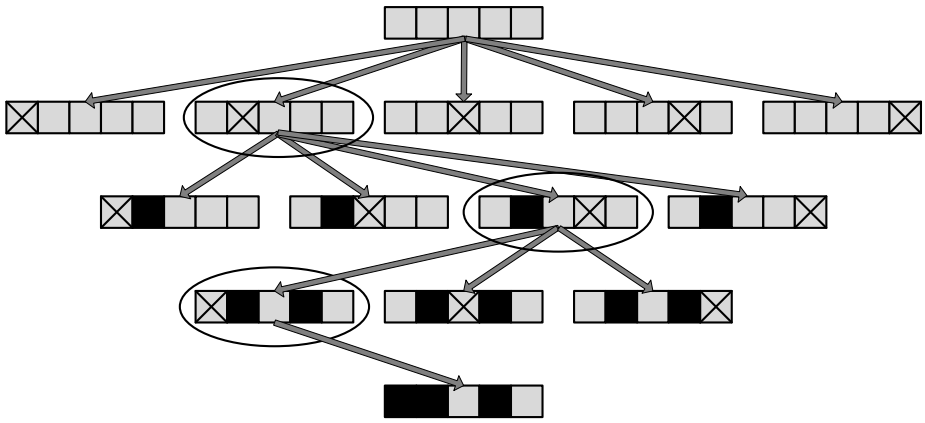


**Figure 4.8 The flip-flop selection process** . The flip-flop whose elimination leads to maximum retention of restored states according to the estimation metric is decided to be removed in next round. The blackened out flip-flops has been already eliminated, while we have to try out all elimination possibilities(shown by crossed) before deciding upon the next elimination. In this example the trace buffer width is 2, so 2 flip-flops are selected out of 5.

The problem of selecting the optimal set of flip-flops can be viewed as a problem of retaining the maximum amount of information in the unrolled circuit graph. We start off with all flip-flops in the circuit (which will restore almost all signals and states), and then we try to constrain this set by removing flip-flops. This will ensure that we do not get constrained by our sub-selections when selecting a larger set of trace signals as pointed out in Section 4.3.1. The flip-flops whose knowledge contribute least to restoring others should get eliminated earlier. When all but the desired number of flip-flops are eliminated, this process terminates. We use the previously proposed simulation based metric, as an estimate of the

information retained by the remaining set of flip-flops. If elimination of two or more candidate flip-flops result in same amount of state restoration in mock simulation, we break the tie by comparing total number of signals restored. If a tie still exists then it is broken by considering the number of other flip-flops, that the candidate flip-flop is connected with via a forward or backward path in the circuit graph. The flip-flop with less connections will get eliminated, if a tie still remains it will be broken by random choice.

**Input:** *circuit*, width of trace buffer $w$, mock simulation based SRR estimator $f_{SRR}(...)$
**Input:** parameters: step-size $d$, pruning termination parameter $PT$
**Output:** selected flip-flop set $T$

1: The set of all flip-flops in the circuit $S$;
2: Current observability $V = f_{SRR}(S) \times |S|$;
3: Start with all flip-flops $T = S$;
4: **while** $V > PT$ **do**
5:  **for** each flip-flop $s$ in $T$ **do**
6:  $T = T - \{s\}$;
7:  Observability $V = f_{SRR}(T) \times |T|$;
8:  Restoration capacity without $s$ $RCW[s] = V$;
9:  $T = T \cup \{s\}$;
10:  **end for**
11:  $T = T - \{s | RCW[s]$ is within top $d$ values $\}$;
12:  $V = f_{SRR}(T) \times |T|$;
13: **end while**
14: **while** $|T| > w$ **do**
15:  Maximum observability $maxV = 0$;
16:  **for** each $s$ in $T$ **do**
17:  $T = T - \{s\}$;
18:  observability $V = f_{SRR}(T) \times |T|$
19:  $T = T \cup \{s\}$
20:  **if** $V > maxV$ **then**
21:  $selected = s$;
22:  $maxV = V$;
23:  **end if**
24:  **end for**
25:  $T = T - \{selected\}$;
26: **end while**

**Figure 4.9 Pseudo-code for the final algorithm.**

This method is shown in Figure 4.8. Note that if we start with $N$ flip-flops, it takes $O(N^2)$ steps to converge at the final set. Hence, for large circuits the this might become very computationally demanding. We noticed that in typical circuits some flip-flops are always restorable from the knowledge of other flip-flops and hence they do not carry any information. We take advantage of this by performing a fast pruning on a large number of

flip-flops, to reduce this size of the set to an extent that application of an $O(N^2)$ algorithm will be feasible. To perform this pruning, we consider the SRR estimate of each possible set by removal of one flip-flop, however instead of only removing the flip-flop whose elimination leads to maximum estimated SRR and repeating the process, we remove a set of flip-flops which have poor information content, in one step. We consider all possible eliminations in sorted order of SRR estimate values (as $RCW[]$ in 4.9). The flip-flops whose elimination lead to the top few SRR estimate values are the candidates to be in the final elimination set. The size of the set is a parameter called step-size $d$. For our experiments this parameter was set as 50. To limit the extent to which this coarse grain pruning is done on a circuit, we can specify a pruning termination parameter $PT$ such that if the average number of restored flip-flops in the mock simulation drops below that value, the coarse grain pruning will stop and the actual elimination algorithm will work on the residual set. This parameter can create a trade-off between quality of selection and the execution performance of the algorithm. It was chosen as 95 percent of the total number of flip-flops in the circuit to assure good quality of signal selection for our experiments. The final algorithm is illustrated in Figure 4.9.

## 4.6 Experimental results

We evaluate the quality of the trace signals selected by the proposed algorithm by comparing SRR obtained on six ISCAS89 benchmark circuits, which were used in previous works that strive to maximize restoration [59, 61, 80, 13]. The number of flip-flops in the circuits and other circuit characteristics are presented in Table 4.1. The benchmarks are re-synthesized using Synopsys Design Compiler targeting the GTECH gate library, to conform with the quality of optimization performed on netlists used in industry currently (re-synthesis is performed in[86] as well). Note that, some redundant flip-flops in these designs are removed by the synthesis tool.

| Circuit | # Flip-flops before synthesis | # Flip-flops after synthesis | # Gates after synthesis |
|---|---|---|---|
| s5378 | 179 | 164 | 1,058 |
| s9234 | 211 | 145 | 920 |
| s15850 | 534 | 524 | 3,619 |
| s38584 | 1,426 | 1,426 | 12,560 |
| s38417 | 1,636 | 1,564 | 10,564 |
| s35932 | 1,728 | 1,728 | 4,981 |

**Table 4.1 Benchmark circuits used to evaluate proposed signal selection algorithm**

The X-simulator which restores the value of non-traced signals and states forms an integral part of our solution since it is used to compute the estimation metric through mock simulations, as well as for measuring SRR attained by the algorithm. The 3-input or larger gates are internally de-composed into elementary 2-input gates in the X-simulator for efficient computation, a transformation that has no other consequence since the trace signals are only flip-flop values. We implemented our X-simulator using the efficient event-driven bit-parallel forward and backward propagation technique described in [59]. All the experiments were run on a quad core Intel processor running at 2.4 GHz. The width of the bit-parallel operations in the restoration process was extended to 64 bits from the 32 bits described in the original, to utilize the 64 bit word size of the processor, which greatly increases the performance of individual mock simulations, performed for a depth of 64 cycles.

During the tracing operation each circuit was kept in the functional mode, by keeping global reset signals de-asserted and forcing fixed values at other control inputs while feeding random values at other primary inputs. This input restriction is referred as "deterministic random" in several previous works [59, 13]. This restriction at the inputs is very important to evaluate the quality of trace signal selection. If control inputs are allowed to toggle, the circuit might intermittently enter the reset state and the reset signal itself might be traced, leading to a large amount of state restoration. However, during debug this scenario is unlikely to happen and the circuit will remain in the functional mode most of the time, so the state restoration ratio obtained when control signals are allowed to toggle is not representative of actual restoration capacity of the trace signals. This issue has been pointed out in [59, 61]. All our experimental results correspond to the circuit operation in functional mode, and all the mock simulation estimates are also obtained under this constraint.

### 4.6.1 Restoration quality

Table 4.2 compares the state restoration ratio obtained by several previous solutions with our proposed technique on the ISCAS89 benchmarks. As in [61, 13], the trace buffer widths used in the experiments are 8,16 and 32, while the depth is kept at 4096 cycles and corresponding SRR for each solution (wherever known) is reported. The percentage improvement of SRR obtained by the proposed algorithm over the best reported value is reported in last column. Each reported restoration ratio for the proposed algorithm is the average over 100 simulations, with 10 different seeds (to generate random values at non-control primary inputs), and 10 different cycle offsets from the initial reset state, per seed.

| Circuit | trace width | Ko & Nicolici [59] | Liu & Xu [61] | Basu & Mishra [13] | Proposed Solution | Improv.(%) over best |
|---|---|---|---|---|---|---|
| s5378 | 8 | - | *14.67* | - | *13.24* | *-9.75* |
| | 16 | - | *8.99* | - | *7.83* | *-12.93* |
| | 32 | - | *4.72* | - | *4.89* | **+3.60** |
| s9234 | 8 | - | *4.76* | - | *10.68* | **+24.36** |
| | 16 | - | *7.18* | - | *7.16* | *-0.27* |
| | 32 | - | *4.67* | - | *4.18* | *-10.49* |
| s15850 | 8 | - | 19.93 | - | 39.54 | **+98.39** |
| | 16 | - | 24.22 | - | 24.85 | **+2.60** |
| | 32 | - | 13.30 | - | 13.60 | **+2.25** |
| s38584 | 8 | 19.00 | 19.23 | 78.00 | 84.10 | **+7.82** |
| | 16 | 10.56 | 13.96 | 40.00 | 47.04 | **+17.60** |
| | 32 | 6.32 | 8.68 | 20.00 | 26.97 | **+34.85** |
| s38417 | 8 | 19.62 | 18.63 | 55.00 | 45.21 | -17.80 |
| | 16 | 11.22 | 18.62 | 29.00 | 30.77 | **+6.10** |
| | 32 | 6.73 | 14.20 | 16.00 | 20.25 | **+26.56** |
| s35932 | 8 | 41.45 | 64.00 | 95.00 | 96.12 | **+1.17** |
| | 16 | 39.31 | 38.13 | 60.00 | 67.45 | **+12.41** |
| | 32 | 24.76 | 21.06 | 35.00 | 43.23 | **+23.51** |

**Table 4.2 State restoration ratio without input knowledge for ISCAS89 circuits**. Only traced state elements are used for restoration. SRR obtained by previous solutions which only use the knowledge of traced signals are presented for comparison. The last column represents percentage change over the best reported in literature.

For certain buffer sizes, especially in the case of smaller sized ISCAS89 circuits the SRR obtained by our solution is less than that of the best reported. This anomalous behavior is primarily caused by the fact that the optimized ISCAS89 circuits have a reduced number of flip-flops. Hence, even though our technique actually restores higher percentage of flip-flops on average per cycle the reported SRR of previous solutions is often boosted by restoration of the redundant flip-flops. As an example, for buffer size of 32 in the case of s9234 circuit, our algorithm restores 4.18x32 = 134(approx.) flip-flops on average per cycle out of 145, which is 92 percent of all flip-flops, where as the best reported solution only restores 4.67x32=149(approx.) out of 211, which is only about 70 percent. For the larger circuits, which are better representative of the cases encountered in post-silicon debug, our solution achieves up to 34.85 percent (for s38584) better state restoration ratio.

### 4.6.2 Effect of pruning

We studied the effect of the pruning optimization (discussed in Section 4.5) on top of our elimination based algorithm. The effect of pruning is shown in Figure 4.10. This data corresponds to execution of the proposed algorithm for circuit s15850, when the $f_{SRR}()$ metric is using a mock simulation of depth 32 (instead of usual 64, for purposes of visible fine granularity), and the trace buffer width is set at 32. Hence the algorithm terminates at trace set size of 32. A total of 524x32=16768 flip-flop values are present in the window of mock simulation (s15850 has 524 flip-flops refer Table 4.1). The y-axis effectively plots the value of $f_{SRR}(T) \times |T| \times 32$ during each iteration in the execution of our signal selection algorithm. Note that the no-pruning line is smooth as only one flip-flop is removed per iteration, and the total number of restored flip-flops in the mock simulation gradually decreases. On the other hand, pruning uses a step-size($d$) of 50 flip-flops, hence during the pruning phase total number of restored flip-flops drop as a step function at each 50 interval. In this example pruning termination($PT$) was set at 93 percent of all flip-flop values i.e. 16768x0.93=15594, by which point the whole set of 524 has already been reduced to around 200. Note that the pruning produces only slightly lesser quality signal selection than exact version, as the with-pruning line ends slightly lower than the no-pruning line. Thus pruning sacrifices accuracy to a small degree for faster execution of signal selection algorithm.

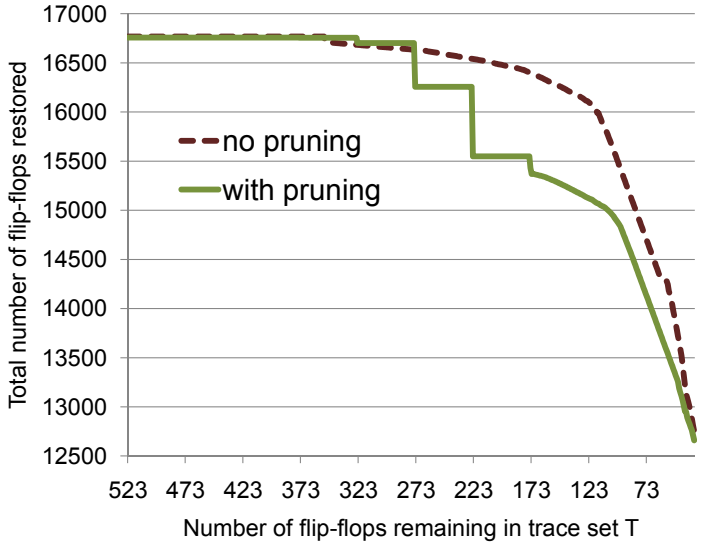



**Figure 4.10 The effect of pruning during execution of trace signal selection algorithm is shown for circuit s15850.**

| Circuit | cpu (s) | gpu (s) | speedup |
|---|---|---|---|
| s38584 | 18,003 | 6,703 | 2.69 |
| s38417 | 24,734 | 8,021 | 3.08 |
| s35932 | 19,801 | 6,501 | 3.05 |

**Table 4.3 GPU acceleration of the selection algorithm**. Execution times corresponding to trace buffer width of 32 is reported for all cases.

### 4.6.3 Algorithm execution performance

We implemented a parallel version of the X-simulation kernel on the GPU, which performs the $|T|$ independent simulations needed in every step of the elimination algorithm, in a parallel fashion. We used a NVIDIA GTX 480 GPU as the execution platform and it was programmed through CUDA [73]. Each distinct thread-block performs the X-simulation using a different traced flip-flop set. The main restoration algorithm was also modified in order to fit single instruction multiple thread execution paradigm used by GPUs. The performance improvement obtained from parallel execution is reported in Table 4.3. Note that performance of the parallel version is comparable or even better compared to other solutions.

## 4.7 Related Work

Automatic trace signal selection algorithms for debug are a fairly new research area. One of the first solutions in this domain [45] considered only the reconstruction of data at the combinational logic nodes of the circuit. Ko and Nicolici [58] defined the term state restoration and introduced an efficient algorithm to perform state restoration as a post-analysis process on recorded trace-buffer data. They also introduced the first trace signal selection algorithm striving to maximize the amount of restored state. Further research in this area has produced several improved solutions for automatic signal selection [61, 80, 13], all sharing the goal of improving the SRR.

As mentioned earlier, these solutions share a common structure, with a metric to estimate the restoration capacity of a certain set of state elements and a greedy selection algorithm to decide which ones to trace, based on the estimator metric. These previous solutions primarily differ in the way estimation is performed. Both [58] and [61] leverage a probabilistic metric: the steady state probability of the value at flip-flop outputs is estimated assuming uniform random distribution of 0 and 1 logic values at the primary inputs. Given these assumptions and using the knowledge of the traced signal values, a probabilistic model of the *visibility* of 0 and 1 values at the other circuit nodes can be generated.

This probabilistic model leverages the circuit topology and logic functionality of individual gates, and the estimation process performs forward and backward propagation of probability values across logic gates. The final state restoration capacity estimate is then expressed as a sum of the predicted visibility of 0 and 1 values at the state elements of the circuit. The probabilistic model presented in [58] lacks theoretical basis and it is then improved on in [61]. In contrast, [13] considers only the restoration probability along paths connecting flip-flops. The probability that a flip-flop output value controls the input value of another flip-flop is computed and called *direct restorability* of the corresponding path. The selection algorithm grows a region of flip-flops in a greedy fashion based on this metric, while an adjustment mechanism accounts for flip-flops that are already selected in the region and updates the path's probabilities accordingly. Another solution presented in [80] estimates the visibility of non-traced nodes by non-trivial logic implications of flip-flop values. However, [80] assumes that in addition to trace signals, all primary input values for every cycle are known to the restoration algorithm. Our proposed solution is fundamentally different from these previous ones as it relies on simulation for estimation instead of a probabilistic metric.

Another line of research [96, 86] suggests that not all state elements or signals are equally relevant for debugging purposes. Hence, instead of striving to maximize the state restoration ratio, the authors of those works focus on maximizing restorability of a specified subset of signals, while minimizing the impact to other flip-flops. In particular, the algorithm in [86] uses a probabilistic estimation metric analogous to [61], and follows a pareto optimal selection process. We show that our solution can be adapted to solve this problem variant as well, by simply assigning larger weight coefficients to the set of critical flip-flops.

## 4.8 Summary

Providing observability for debugging beyond software-based simulation is one of the biggest challenges. Lack of observability often renders the simulation performance of hardware-accelerated platforms futile towards debugging. Only a small number of signals can be recorded in these platforms for various performance reasons. However, observability can be achieved via reconstruction of signals. State restoration ratio is a measure of success in terms of restoration. Solutions in this space attempt to devise algorithms that select signals which will lead to maximum state restoration. We presented a trace signal selection algorithm that strives to achieve this objective and therefore paves the way to better

debugging. The selection algorithm is guided by a more accurate simulation based restoration capacity metric and achieves better state restoration ratio than previous solutions. It also achieves better trends of restoration per additional traced signal while restoring higher average number of states. Overall, this solution provides a higher degree of observability into the design for debugging purposes via restoration, than previous solutions. This chapter concentrated on efforts to obtain observability for design debugging capabilities on hardware-accelerated simulation platforms. The next two chapters will explore solutions to bring in design checking capabilities to these platforms.

# Chapter 5

# Providing Checking Capability for Hardware-accelerated Simulation

Checking design behavior for functional correctness is one of the most critical components of simulation-based design validation. To its credit, software-based simulation provides a feature-rich environment for checking needs, which is critical towards validating and debugging a design. A plethora of solutions are available for simulation-based validation of digital designs using software-based simulation [94]. Design behavioral checkers are almost exclusively crafted in high-level functional languages or special purpose verification languages, and are an important part of a testbench. In a typical setup, a large number of complex software checkers are used to validate different components of the design under verification (DUV); these interface with the design and are executed concurrently during the simulation. This powerful checking capability is rather straightforward in software-based simulation, since both the design and the testbench components (including the checkers) are seamlessly integrated in the simulation software. However, the performance of software-based simulation is far short of adequate in practice, since it is typical in these setups to execute approximately one to ten clock cycle of the simulated design per second (1-10Hz): since the final design typically executes at Gigahertz frequencies, it is evident how that performance does not allow for adequate exploration of the design's behavior. As a result, the industry has started to shift more and more of the validation effort towards hardware-accelerated simulation platforms.

Unfortunately, as discussed in Chapter 2, as we move towards hardware-accelerated simulation, checking capability diminishes severely. This problem is primarily due to the fact that these platforms are only designed to carry out high-performance simulation of synthesized digital logic, and do not provide capabilities for checking constructs. Hence, checker-centric validation, although very successful for software-based simulation, fails to extend to the realm of acceleration or emulation. The advantage of simulation performance in hardware-accelerated platforms is often rendered futile for verification purposes due to the lack of checking capability. Thus, to fully unlock the potential of hardware-

accelerated platforms, it is critical to develop solutions that bring checking capabilities to such platforms and enable high-performance simulation-based validation. This chapter of my dissertation develops novel solutions to tackle this challenge and brings checking capabilities to hardware accelerated platforms. The approaches described in this chapter should enable practitioners to perform high quality verification on hardware-accelerated platforms while enjoying their performance advantage. While these solutions focus on microprocessor designs and the case studies target acceleration platforms, the fundamental concepts are applicable to other types of designs, as well as other types of hardware-accelerated simulation platforms.

Section 5.1 provides a background study about current solutions for providing checking capabilities on hardware-accelerated simulation platforms, and highlights the associated challenges. Section 5.2 presents an overview of the solutions, and the subsequent sections describe these solutions in detail. Related work is presented in Section 5.12 and finally Section 5.13 concludes this chapter.

## 5.1 Background

A simulation-based validation environment commonly involves checkers that are connected to the design. These checkers are written in high-level languages, such as C/C++, SystemVerilog, and interface with the design via a testbench. Though simulation-based checking solutions come in several flavors, there are two main fundamental types of checking solutions. We can express the subset of correct behavior of a design as an assertion, which can then be checked during simulation to detect any incorrect behavior as an assertion failure. This is known as assertion-based verification (ABV). Another approach entails connecting a golden functional model to the design under verification (DUV). Often the golden model is developed at a much higher level of abstraction and using a high level functional language such as C/C++, or special purpose verification languages such as Vera or Specman's *e*. A software checker consists of such a golden model along with checks that determine whether the DUV and the golden model outputs agree with each other. Typically, a number of such software checkers are attached to different blocks of a complex design, such as a microprocessor design. Commonly used hardware description languages (HDL) offer ways to interface the software-checker with a design through programmable interfaces, for example Verilog VPI, PLI and SystemVerilog DPI. Special purpose verification languages also provide such capability. The tight coupling between the checker functions and the simulated design allows for a relatively low-effort checker

design. However, such excellent checking capability is crippled by the poor performance of software-based simulation.

As explained in Chapter 1 and 2, a number of hardware-accelerated simulation platforms are becoming increasingly important for performing simulation-based validation. These platforms include acceleration, emulation and prototyping platforms. A large class of acceleration platforms are composed of large arrays of customized ASIC processors [30, 23], specifically designed to simulate logic gates concurrently. To target these platforms, the design under verification (DUV) must be synthesized into a structural netlist, and then the corresponding logic gates are mapped to the execution substrate. Field-programmable gate arrays (FPGA) are the building blocks of emulation and prototyping platforms [10, 67]. FPGAs consist of lookup tables (LUTs) that can be programmed to emulate finite sized partitions of logic. The design's logic is mapped to these lookup tables. Clearly, these platforms are designed for high-performance logic simulation, but are incapable of executing complex checking constructs. The one feature they provide to support checking capabilities is in allowing the recording of a pre-specified subset of design signals. Generally, these platforms are attached to a host computer from which the simulation process is controlled and to which the recorded data is transferred.

In current industry practices, the testbench is stored and executed on the host computer and controls the simulation running remotely on the platform. Selected signals are logged on the platform itself and periodically off-loaded to the host where they are checked by a number of host-bound software checkers to establish the functional correctness of the simulated design. Transfer bandwidth to and from the host can be much smaller than that to support the transferring of data generated for the target checking activity. Hence, often, the logging and off-loading activities become the performance bottleneck of the entire simulation [67, 56]. The whole process can become very inefficient, failing to leverage the performance advantage of the platform. Lock-step execution of software checkers in the host is also not feasible, since it would require stalling the execution and transferring relevant values from the platform, at each simulation cycle; this would hinder performance unacceptably. Moreover, any solution that attempts to provide efficient checking capabilities for hardware-accelerated platforms must be aware of the constraints inherent to these platforms. Two of the most important constraints are discussed below.

**Limitations on logic size:** ASIC-based acceleration platforms may not have a strict logic capacity limit; however, they experience a performance penalty when increasing the amount of simulated logic. FPGA-based emulation / prototyping platforms have strict logic limits dictated by the amount of lookup tables available on such platforms. This logic capacity limit prohibits the mapping of any arbitrary checking solution into equiva-

lent hardware (even if it were possible to do so) to simulate alongside the design. When additional logic is used for checking purposes alongside the design, the associated footprint must be within the bounds imposed by the specific platform.

**Limitations on recording signals:** All hardware-accelerated simulation platforms allow for the collection and transfer of design signal values for checking/debugging purposes, but the transfer slows down the simulation, eroding the key benefit of acceleration. In general, the more signals are observed and transferred, the lower the acceleration performance. However, the precise relation between acceleration performance impact and number of signals traced depends on the specific architecture of the accelerator. Containing the number of recorded signals per cycle (thus the traced data generation rate) is extremely important for a successful checking solution for acceleration platforms. This is due to the fact that the underlying architecture of the acceleration platform records the values of the signals marked for observation in each cycle and stores them in internal memory. Every time the buffers become full, the simulation must be temporarily suspended to transfer the content via a low bandwidth channel to the connected host machine. The more frequently this event takes place, the higher the associated performance penalty. Thus, the lower the number of traced bits, the longer it takes to exhaust the internal buffer resources, and the longer the intervals of uninterrupted simulation. Emulation platforms have very similar trade-offs as well.

## 5.2 Towards providing checking capability

In view of the constraints described in the previous section, providing checking capability in hardware-accelerated simulation platforms is a challenging problem. Researchers have investigated several possible directions to enable such checking capability. One such direction is to **map existing software checkers to hardware descriptions**. Indeed, if we could convert existing software checkers to equivalent synthesizable hardware descriptions, we could simulate them alongside the design. Prior research has investigated synthesis of formal temporal logic assertions into synthesizable logic [2, 31], targeting those platforms. Techniques for using reconfigurable structures for assertion checkers, transaction identifiers, triggers and event counters in silicon have also been explored [3]. However, synthesizing all checkers to logic is often not viable for multiple reasons. Though these checkers can be translated into temporal logic assertions and subsequently synthesized with tools such as those described in [2, 20], the size of the generated logic is often prohibitive. Indeed, the logic size of a checker implementing a golden model for a microarchitectural

block is often as large as the block itself, and such vast overhead is not tolerable.

**Approximate checking:** This chapter introduces a solution called **"approximate checking"** to solve the aforementioned problem. An ideal embedded checker should be sufficiently small not to impact significantly the performance of simulation, while it should be functionally sophisticated enough to contribute to design-level correctness checking (in contrast with a simple local logic check), and thus would be a good substitute for a software checker. Approximate checkers fulfill these requirements by being small enough to not impact simulation performance, yet capable of detecting a significant fraction, if not all, manifestations of a bug in the design (indeed, even just one detection of a given bug is sufficient to expose it). This is achieved by either relaxing or further restricting the checking function of the software checker so that its hardware version becomes deployable with a tolerable logic footprint. This solution provides guidelines on how to approximate different classes of checkers and enables a large variety of software checkers to be adapted for hardware-accelerated simulation platforms. As a result of this transformation we are essentially trading off checking accuracy with logic footprint. Section 5.3 to Section 5.7 details this solution. Our experimental results demonstrate that a large reduction in logic footprint can be achieved at a minor loss of checking accuracy.

Another direction in this space is a **"log and then check"** approach. In this approach, a number of signals relevant to a particular checker is logged during simulation, and this log is checked offline for correctness by a software checker post-simulation. This approach is able to tackle those checkers that are too complex to be translated to equivalent hardware. As mentioned before, hardware-accelerated simulation platforms allow tracing signal values; however, there is an increasing performance penalty with the number of recorded signals. Hence, the challenge in this approach is to minimize the volume of logged data to maintain platform performance and yet not lose checking intent or accuracy. Finally if additional logic is required to perform tracing, we must ensure that it does not slow down simulation performance as well.

**On-platform compression:** An additional solution is presented in this chapter to make the "log and then check" approach effective within the platform constraints. The fundamental idea behind this solution is to reduce the volume of the logged data (which ultimately relates to the number of traced signals) for a particular checker by performing **on-platform compression** of the associated data. It is important to choose compression schemes that are sensitive to design behavior discrepancies, to maintain the same level of checking accuracy as the original software checker, while the amount of logic necessary to perform such compression should be minimal. Moreover, we can further reduce the volume of traced data by not choosing to log design behavior information that can be reconstructed or in-

ferred. This solution is demonstrated for an Instruction-By-Instruction (IBI) architectural behavior checking scheme for an industry-scale microprocessor design on an acceleration platform. For this particular case study, the register value data necessary for checking is compressed on-platform using minimal overhead parity-checksum schemes. Sections 5.8 to 5.11 present the details of this solution. This study demonstrates that it is possible to retain the same level of checking accuracy with only minimal loss of simulation performance due to recording signals.

## 5.3 Reducing checker logic overhead with approximation

This section delineates an approach to map software checkers, traditionally used in software-based simulation, directly to the acceleration platform. We envision that checkers are synthesized and embedded in the acceleration platform, so that both logging and off-loading can be eliminated and data transfer between host and acceleration platform are minimized. However, mapping complex checkers, such as golden models or checkers making use of complex software data structures, remains a challenge because (i) embedded checkers can only use synthesizable constructs, (ii) the logic complexity of their hardware counterparts should not exceed the platform capacity and (iii) the performance impact incurred in the simulation of the hardware-mapped checkers should not cancel out the performance benefits gained by eliminating software-bound checkers.

In this work, we address the problem of designing checkers for simulation acceleration. Our primary objective is to capture the design intent of a complex software checker into a hardware version that can be mapped along with the design to the acceleration platform. The hardware checker must entail a small logic overhead and provide similar capabilities than the original one, but may be approximated. Our proposed solutions trade-off checker accuracy with logic complexity. We provide a classification of common types of checkers (Section 5.4) and then discuss approximation techniques that can be deployed for each type (Section 5.5).

The approximation process may lead to the occurrence of false positives, false negatives and/or delays in bug detection. To properly analyze these effects, I provide metrics to analytically evaluate the quality of an approximation (Section 5.6), and present a case study to demonstrate these concepts (Section 5.7). Our results indicate that we can achieve a reduction of approximately 60% in overall logic complexity at a nominal checker accuracy cost.

## 5.4 Checker classification

Our experience with various designs seems to suggest that even though there are a myriad of checks to be performed by a single verification environment, most fall under one or more of the following main classes, based on the design properties they intend to verify.

**Protocol Checkers:** verify whether the DUV interfaces adhere to the protocol specifications. A checker that checks the request-grant behavior for a bus arbiter is an example. It may check that the arbiter is setting the grant signal no later than a fixed number of cycles after receiving a request, as per the specification. It may also check that the arbiter never issues a grant when some other requester has the bus. A protocol checker may keep track of the expected internal state of the DUV and use it to infer correct behavior.

**Control Path Checkers:** verify whether the flow of data within the design is proceeding as intended. An example control path checker is one that monitors input and output ports of a router to check whether or not a packet that is accepted by the router is eventually sent out through the correct output port. Control path checkers need to keep track of data items for extended periods of time as they are transferred, thus requiring significant storage. They must also mimic the DUV actions applied to the data to determine if the correct transformations are being applied.

**Datapath Checkers:** verify whether data is being manipulated as expected. A datapath checker for a processor's ALU, for example, verifies that the result of an addition operation is actually the sum of the operands. In a software checker, verifying computation is as easy as simply describing the computation in a few statements and comparing the result with the output from the DUV. Implementing a datapath checker in hardware requires a full-fledged functional unit to compute the desired result. Where possible the unit could be simple, targeting only functionality rather than also performance or power. For instance, a simple ripple-carry adder design could be sufficient to check additions.

**Persistence Checkers:** verify whether or not data stored inside the DUV become corrupted. A checker verifying the contents of a processor's register file is an example. It could check that the contents of a register never change unless acted upon by a write command. Much like the checkers seen before, persistence checkers require some information to be maintained about the internal state of the DUV – the contents of the register file in our example. Thus, an embedded hardware version may include a duplicate storage unit.

**Priority Checkers:** verify whether specified priority rules are being respected. A priority rule sets the order in which certain operations are to be performed, usually selected from some type of queue. Consider a unified reservation station in an out-of-order processor that must give priority to addition operations going to the ALU over shift operations. A

priority checker for this unit must verify that no shift operation is issued when there are additions waiting. Embedded versions of priority checkers typically do not reflect their software counterpart structure, because of the challenge of mapping it; instead they are often structured as a combinational logic block monitoring internal design signals.

**Occupancy Checkers:** verify that buffers in the system do not experience over- or underflow. Expanding on the reservation station example, an occupancy checker may verify whether the processor dispatches instructions into the reservation station as long as there is space available. Similarly, it should check that no dispatch should be possible when the reservation station is full. The hardware structure that keeps track of the necessary information can be as simple as a counter associated with the buffer.

**Existence Checkers:** verify whether an item is present in a storage unit. In a processor cache, for example, an existence checker has to verify that a tag actually exists in the tag array. Existence checkers must track what type of information is written in the storage unit. A hardware counterpart must store sufficient information to determine the presence and type of data in the storage.

## 5.5 Approximation techniques

As we mentioned earlier, to eliminate the performance bottleneck in simulation acceleration due to data transfers between host and acceleration platform, checkers must be embedded in the digital design to be mapped onto the platform. However, a direct translation of a software checker, whenever possible, often leads to an extremely complex circuit block, possibly as large or larger than the design itself. Based on the classification in the previous section, we developed a number of approximation techniques, presented below, to address the issue of checker complexity.

**Boolean Approximation:** A Boolean approximation can be used to reduce the complexity of any combinational logic block. The don't care set of the Boolean function implemented by the block can be augmented by simply changing some output combinations from 1 or 0 to don't care (indicated by X). By selecting appropriately which combinations become don't cares it is possible to greatly reduce the number of gates required to implement the function. An example is shown in Figure 5.1, where two input combinations of the original function are modified to don't care (highlighted by hashing). Because of the change, a sum-of-product implementation of the function goes from 6 gates to 4 gates.

Boolean approximation often allows great reductions in circuit complexity with a minimal amount of don't care insertions. Note also that the transformation may lead to false

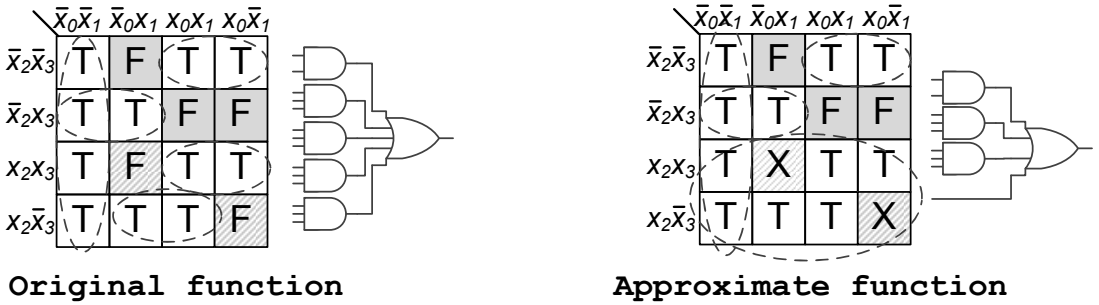



**Figure 5.1 Boolean approximation for a four input function.** Replacing some output combinations with don't cares reduces SOP logic.

positives or negatives for the checker: a 0 approximated to a 1 would lead to a false positive, and viceversa. However, it is possible to somewhat control the incidence of false detections by only transforming input combinations that occur infrequently. Note that for a general checker implemented in sequential logic, it is still possible to apply the technique by first unrolling the circuit for a fixed number of times and then applying the technique on the resulting combinational function. Finally, when the checker (particularly the unrolled checker) has a large number of input variables, this approach may be difficult to apply, and even harder to tightly control its false detection rate: we plan to investigate these situations further in our future work.

**State Reduction:** Embedded checkers may include storage elements for a wide variety of purposes. State reduction eliminates some of the non-critical storage to simplify both the amount of sequential state and the corresponding combinational logic. An example of non-critical storage are counters to check timing requirements of events – counters may still be available, but have smaller size and only count at larger granularity. In this case the approximation may affect delay measurements. For instance, Figure 5.2 shows a portion of the FSM for a protocol checker that verifies whether or not a signal is set for only one cycle. The extra delay state can be removed and the check can be performed in all the states following the NEXT state. Even though the checker can no longer check precisely the one cycle delay, it can still verify that the signal does not remain high after a bounded number of cycles.

If the checker is connected to a reference model for the correct protocol behavior, this approximation technique may be particularly valuable. Indeed, often the reference model's response must arrive before the design's response in order for the check to operate properly. The approximation would allow the checker to update the reference model and obtain its response before the design's response.

State reduction usually weakens the checker's capabilities and may introduce false detections, either positive or negative.


**Figure 5.2 Portion of an FSM for a protocol checker.** The delay state is needed only for checking strict timing, which can at times be relaxed.

**Sampling and Signatures:** The width of a datapath affects many aspects of a design, including the width of functional units' operands and of storage elements. To contain the amount of combinational logic and storage required to handle wide data, an approximate checker can operate either with a subset of the data (sampling) or a smaller size representation (signature) obtained from the data. Bit-fields, cryptographic hashes, checksums, and probabilistic data structures as proposed in [17] can be utilized for signature-based approximations, trading storage size for signature computation. A golden model for an IPv4 router design, for example, does not need to store all the data bytes of packets entering the system. In most cases, storing just the 16 bit header checksum, source address, and destination address along with a simple XOR signature of the data field, can suffice for checking purposes (see schematic in Figure 5.3).

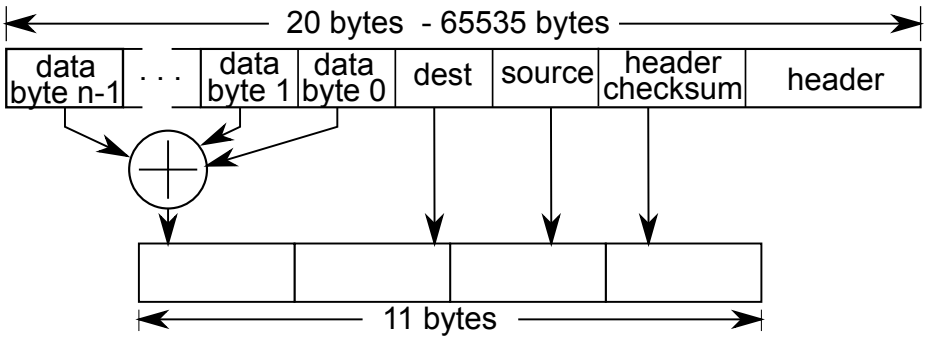



**Figure 5.3 Approximate representation for an IPv4 packet**. Only a few bytes can identify a packet uniquely with high probability.

Depending on how the sampled/signature data is to be used, this approximation may result in both false positives and false negatives. As we will show in a subsequent section, one can reasonably estimate the effect of these techniques, given the probability distribution of the data payloads and the nature of the design.

We considered all the checker types in our collection of checkers from academic and industrial designs and studied which approximations can be applied to the various types of checkers. The results of our analysis are shown in Table 5.1. Note that Boolean approximation and state reduction are general techniques and applicable to all the common types. However sampling and signatures have more limited scope as they are only appropriate for situations where checking on a subset of possible events/combinations can lead to a

detection.

| | Boolean | State Reduction | Sampling | Signature |
|---|---|---|---|---|
| **Protocol** | Y | Y | | |
| **Control Path** | Y | Y | Y | Y |
| **Datapath** | Y | Y | Y | Y |
| **Persistence** | Y | Y | Y | Y |
| **Priority** | Y | Y | Y | Y |
| **Occupancy** | Y | Y | | |
| **Existence** | Y | Y | | Y |

**Table 5.1 Approximation ideas for the checker classes**. Boolean approximation and State reduction are generic methods applicable to any type.

## 5.6 Approximation quality metrics

Approximate checkers may be more relaxed or more restrictive than their original unapproximated counterpart. Thus, depending on the time and ways of a bug manifestation, detection may occur as in the original checker (*true positive or negative*), or the bug may be missed by the approximate checker only (*false negative*), the approximate checker may falsely flag the occurrence of a bug (*false positive*). In this context, it is important to evaluate the relative detection capability of an approximate checker with respect to the original checker. A good approximate checker should have a small rate of false positives and negatives. If the post-simulation diagnostic methodology is capable of ruling out false positives, than a higher false positive rate would not be a critical issue for the approximate checker. We propose to evaluate the quality of an approximate checker with accuracy and sensitivity: two common statistical metrics to evaluate binary classification tests [29].

**Accuracy** measures how faithful an approximate embedded checker is in mimicking of the original software checker. A high value of accuracy indicates that the approximate checker provides accurate detection most of the time (few false positives and negatives). Our accuracy model assumes that each testbench stops whenever a bug is detected (whether that is a correct detection or a false positive) or when the test completes (in case of true or false negative). Below we provide an equation for the accuracy metric that we will use for our case studies in the next section.

$$accuracy = \frac{true\ positives + true\ negatives}{total\ number\ of\ tests}$$

**Sensitivity** tells us how good is the approximate checker in detecting actual bugs (true positive rate). A high value of sensitivity indicates that most bugs can not escape detection by

the approximate checker.

$$sensitivity = \frac{true\ positives}{true\ positives + false\ negatives}$$

When interpreting accuracy and sensitivity many additional aspects must be taken into account, including the input test vectors, the type of the design under verification, the type of checker class being approximated, and the nature of the design bugs must be taken into account. The next section presents two case studies to illustrate these factors and their impact.

## 5.7 Case study: calculator design

I will use a research purpose design to illustrate the concept of checker approximation. It is a calculator design, similar in principle to a microprocessor with a restricted instruction set. Even though the design is small than industrial size designs, it contains enough properties to be verified using checkers that span over the classes discussed earlier.

Calculator 3 aka `calc3` is used as an example in [94]. The design accepts commands to *add*, *subtract*, *shift-left*, *shift-right*, *branch-if-equal*, *branch-if-zero*, *load-register* and *fetch-register* through 4 command ports and responds with results through 4 corresponding response ports. All commands operate using 16 32-bit wide internal registers, shared among all command ports. The *load-register* and *fetch-register* commands, respectively, write and read 32 bit data values to and from the register file. The arithmetic commands (*add / subtract / shift*) take two registers as operands and place the result in a third. The *branch* command compares a register for equality, either with another register or with zero, and sets a branch condition if the test succeeds. A successful branch makes `calc3` skip one following command from the same input port.

According to the specification, the design should support up to 4 pending commands per port and out-of-order completion of commands as long as there are no data hazards. Each command is associated with a unique 2-bit tag value, reported when the command completes, along with 2 status bits indicating a successful completion, a skipped command, or an overflow from addition/subtraction. Only the *fetch-register* command creates an output on the data line for a response port.

The baseline black-box checkers for `calc3` were created by manually translating a high-level C++ software testbench into a Verilog description. Each port has a separate black-box checker ensemble working on the context of a common shadow register file,

which maintains a copy of the values that should be in `calc3`'s register file. The main components of the checkers for each port are shown in Figure 5.4 and they fall within four major classes from Section 5.4:

**Protocol checker.** Does an issued command have an unused tag? Does a completing command have a tag used by a pending command? Is the issued command legal? Do pending commands still remain even after sufficient cycles have passed from the last issue?

**Control path checker.** This checker should validate the following aspects: are commands following taken branches correctly skipped? If an error condition is flagged, is the writing to the result register bypassed?

**Datapath checker.** Aspects to check are: is the correct condition flagged during completion of a command? Is the result of a command correctly computed? Does a fetched register value match its expected value? Is a branch condition correctly recorded?

**Priority checker.** This checker checks that commands completing out-of-order do not violate data hazard constraints.

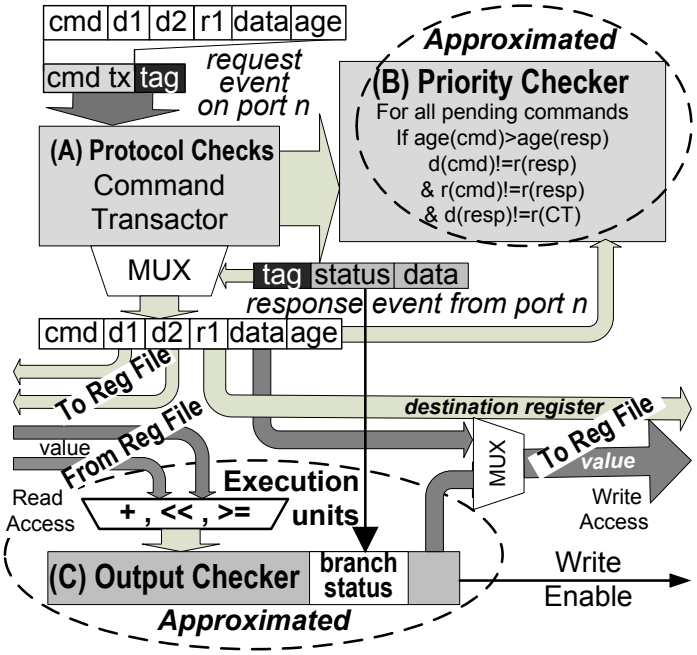



**Figure 5.4 `Calc3` checker ensemble for one port**. Tracking is done as follows: Tags by the command transactor, dependencies by the priority checker, and required computations by the execution units. Results of execution units are used by the output checkers for verifying the correctness of the status bits. Note that since `calc3` does not output the results of an arithmetic operation, the only way for the black-box checker to obtain these results for verification is through a *fetch-register* command.

The main technique used for approximation in this case study was sampling. The output checker and duplicate execution units were approximated by sampling a subset of the 32 bits for each operand. This reduces the amount of logic in the checkers, while it retains

strong checking ability to detect all the control path bugs. The approximated datapath for the output checker operates on the least significant 8 bits of data, except the comparator, which uses all 32 bits. This exception was necessary to ensure the branch decisions do not differ between the checker and the design, which would introduce unnecessary false positives. However sampling leads to logic savings in arithmetic-heavy execution units. Note that this scheme cannot detect overflows; hence the approximate checker relies on the command's status bits to learn about overflows and could potentially miss related bugs (false negatives).

The priority checker is also amenable to sampling, where the completing command is checked for priority violations only with respect to one pending command instead of all of them. This reduces the logic needed to implement the checker to approximately one fourth. Even though this is a weaker check, since incoming commands are mapped to different slots in the transactor based on current occupancy, there is a non-zero probability that a violating command is present in the slot being checked for violation. Hence, with sufficient simulation runs, a bug causing priority violations will be detected. The command transactor could not be approximated since the other checkers depend on protocol adherence and even a slight approximation for this checker would introduce many false positives.

A significant amount of logic reduction can also be achieved by taking advantage of the fact that, unlike the software version, the checker implementation resides in hardware, together with the DUV. Duplication of logic can be limited since additional wire connections can be made to the DUV's components. For instance, we can avoid maintaining a shadow register file by simply checking dynamically that the values to be written are a match with those computed by the checker. With this optimization, the shadow register file can be replaced by shared read ports with the design's internal register file and a register-write checker that checks the least significant 8 bits of written register values.

### 5.7.1 Evaluation of the approximate calc3 checkers

In our evaluation of checker approximation for the calculator design, we injected a number of different bugs into `calc3`. We created several variants of the design: for each bug, we created two versions that included only that bug. The two versions differed only in that one included a complete hardware version of all the checkers described in Section 5.7, and the other included the approximate checker(s) instead. The bugs varied widely in their complexity and the types of checkers they triggered. Each simulation could terminate either because a bug was detected or because the test run to completion. Each bug detection (or

lack thereof) by an approximate checker was compared to the corresponding detection by its complete counter-part and then labeled as a true positive, true negative, a false positive or false negative.

| id | checker | description |
|---|---|---|
| adds | cmd tx | only dispatch adds when shift and add commands can be dispatched |
| ovr | output | add or subtract with overflow writes register |
| stuck | output | 20th bit in register 13 is stuck |
| stall | cmd tx | 11th add/shift/branch command stalled |
| blk1 | output | second branch with same tag not blocked |
| dreg | output | dispatches an add and shift to same dest. register at same time |
| blk2 | priority | command with tag 11 is not blocked by command with tag 00 |
| iraw | output | an incoming command reads a register being written in same cycle |
| eraw | output | an enqueued command reads a register being written in same cycle |
| skip | output | branch follower not skipped following branch |

**Table 5.2 List of bugs for the `calc3` design**. The checker field shows the unit that detects the bug. In the approximated version, all the bugs shown to be detected by the output checker are seen as register write mismatches.

The bugs injected in the `calc3` design are described in Table 5.2. We ran a total of 500 tests on each design variant obtained by injecting a different bug. In most cases, randomly generated command sequences were adequate to sensitize the bugs. For the few hard-to-sensitize bugs, we inserted control command sequence snippets at random times in the simulation.

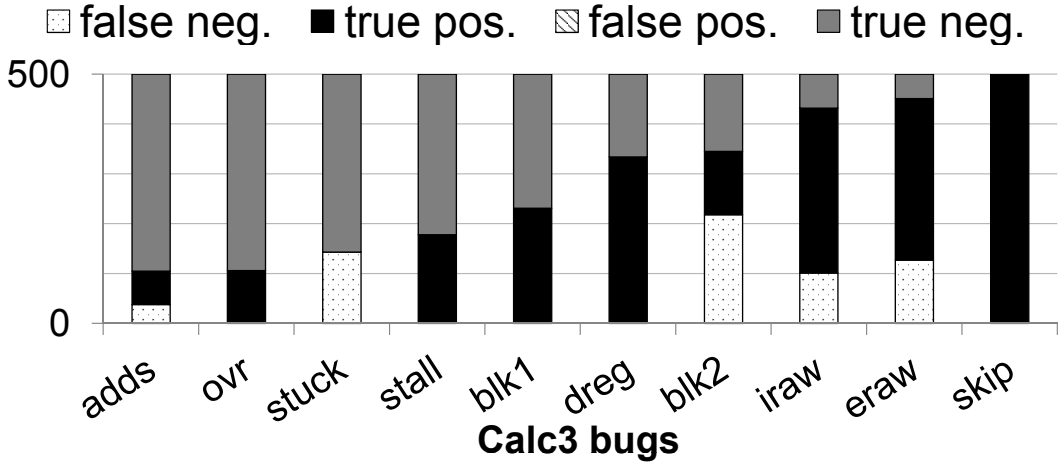



**Figure 5.5 Distribution of detections for `calc3` bugs**. `Calc3` approximations were designed to avoid false positives.

Figure 5.5 shows the breakdown of the test outcomes: for each bug, we report how many tests resulted in each outcome type. Note that for the *stuck* bug, corresponding to a stuck bit in register 13, our approximation scheme was not able to detect any occurrence, since the stuck bit position is not within the range of bits monitored by the approximate checkers. As it can be noted from the Figure, for half of the bugs our checkers always

detect the bugs correctly, either true positive or true negative. For the other half we experience some false outcomes, however there is still a significant rate of true positive detections, which allows for proper bug diagnosis. After all the approximations are applied the `calc3` design achieves 87.5% accuracy and 74.9% sensitivity.

Finally, we evaluated the logic complexity of the original embedded hardware checkers and compared against our approximate checkers. To this end, we synthesized all the case study designs and checker variants using Synopsys' Design Complier, targeting the technology-independent GTECH library. Since the process of mapping a digital design onto an acceleration platform is very specific to the platform being used, we simply take the total number of logic blocks generated as an approximate indicator of the size of the design on the platform. Table 5.3 shows the results of our analysis.

| **unit** | **technique** | **original** (#blocks) | **approximate** (#blocks) | **reduction** (%) |
|---|---|---|---|---|
| `calc3` output | sampling | 4,810 | 1,332 | 68.1 |
| `calc3` priority | sampling | 2,928 | 782 | 73.3 |
| `calc3` reg file | eliminate | 7,945 | 1,031 | 87 |
| **`calc3` checker** | **combined** | **20,473** | **8,565** | **58.2** |

**Table 5.3 Logic reduction for `calc3` checker** . Overall checker overhead with respect to the `calc3` reduced from 87% to 36%.

The case study suggests that approximation of checkers is a viable solution that reduces the hardware overhead of complex checkers while still enabling a large fraction of design bugs to be caught.

## 5.8 Leveraging on-platform compression for checking

The previous few sections presented a solution to leverage embedded logic to perform checking on hardware-accelerated platforms. In the following sections, we present an architectural checking solution for microprocessor cores on acceleration platforms that uses the alternate approach of "log-and-then-check". Our solution performs "instruction-by-instruction" (IBI) checking, that is, it validates the outcome of each instruction completed by a processor design in accelerated simulation by comparing it with an architectural golden model. We achieve our goal by applying a number of major transformations to a baseline software simulation-based validation methodology. The IBI checker for acceleration platforms exhibits one of the key challenges is adapting any software checker to a hardware-accelerated simulation platform via the "log and then check" approach: namely, the volume of traced data necessary for checking is too high to retain simulation performance. This problem is solved by leveraging low-overhead on-platform logic to compress

the relevant data and produce a summary, instead of recording raw signals. The summary is then checked off-platform for discrepancies. This novel technique reduces the number of recorded signals heavily, and is able to retain simulation performance. The proposed solution retains almost all the capabilities of its software counterpart but does not compromise the performance of acceleration. We successfully deployed and evaluated this solution in the validation of an upcoming IBM POWER processor design.

### 5.8.1 IBI background

Instruction by instruction (IBI) checking, or golden model based validation, is a well known checking technique that has been used in processor verification for many years [62, 93]. IBI compares the architectural events produced by each executed instruction with those required by the processor specification. This technique provides a simple way to distinguish deviations from the desired behavior. It does not depend on the internal implementation of the processor, and can be used with any microarchitecture implementing the same instruction set. An additional benefit of this approach is the relative ease of debugging: the corresponding checker recognizes the exact spot of the deviation in time and thus it enables the time localization of the problem.

A typical IBI checking methodology works as follows. A test generator (*e.g.* [41, 4]) produces a test program containing the results expected by the processor specification after each instruction (the expected results). These results are usually obtained using a software that can calculate the expected results after each instruction, known as a golden model. Then the checker environment compares these results to the ones produced by the processor simulator for the same test program [62, 93]. The checker environment needs to identify when an instruction execution completes and what resources were modified because of the instruction execution. It also needs to account for the behavior that cannot be predicted by the golden model (*e.g.* external interrupts), or are not fully defined by the specification (*e.g.* values of some registers become "undefined" when exceptions occur).

### 5.8.2 IBI for acceleration platforms

In this section we present our instruction-by-instruction checking solution for acceleration platforms. Our technique enables this validation methodology on fast accelerated simulations, thus boosting the amount of simulation cycles that can be checked within a given amount of time. In our solution, we run the same test on the processor model simulated in the acceleration platform and on the golden model running on the off-platform host,

and then compare results. To make the comparison possible, we need to collect relevant information about the retired instructions and architectural resources modified from the acceleration platform, and transfer it off-platform. The actual comparison is then performed by a dedicated software checker, capable of running the golden model on the same test and compares the two sets of results. As mentioned earlier, the acceleration advantage decreases when increasing the amount of recorded information and the size of simulated logic. Hence, one of our design goals is to record as little information as possible and incur as little hardware overhead as possible, all while delivering accurate bug detection capabilities.

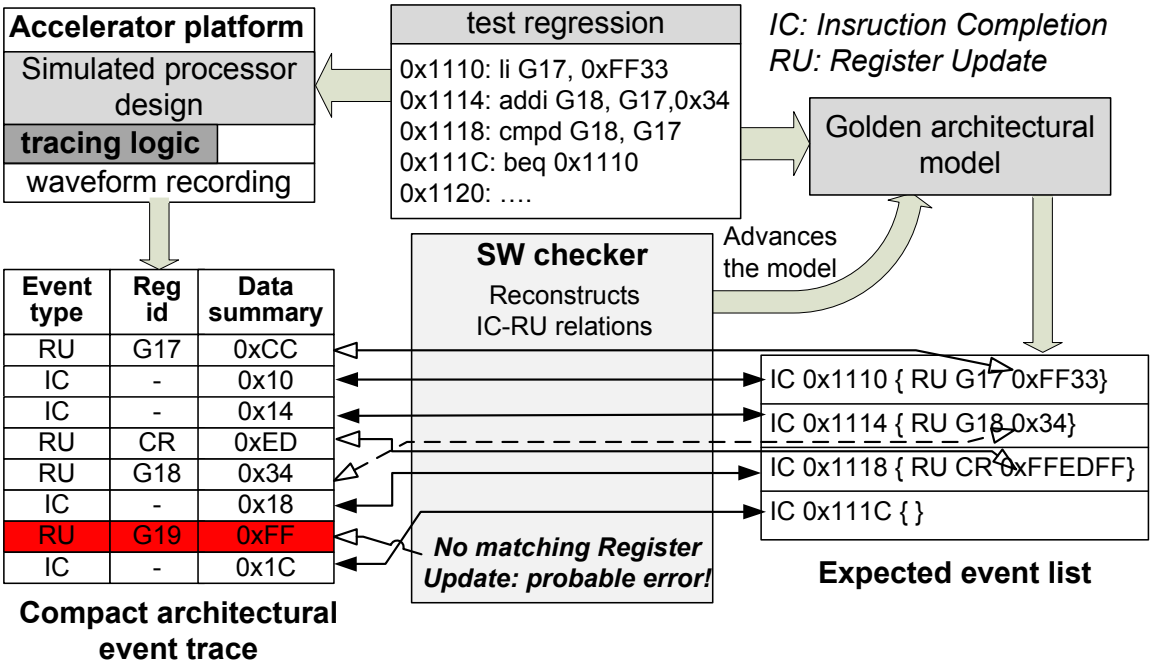



**Figure 5.6 Overview of our solution** to provide IBI checking on acceleration platforms. The Figure illustrates the test running on the platform (left) and on the off-platform software (right). The bottom left table shows an example of data transferred off-platform.

Based on the observations above, our solution comprises the following two components: i) a dedicated, on-platform, logic block to record a compact summary of architectural events and ii) an off-platform software checker module that considers the recorded data and analyzes it in light of a golden model output. This decoupled approach enables us to get around one of the fundamental challenges discussed previously, minimizing on-platform logic overhead. However, it also imposes a substantial redesign of the checking approach. We will check instruction completions and registers only (similar to many other IBI solutions) because memory behavior is very difficult to trace and predict in modern architectures. To achieve this we will record two types of events on the acceleration platform - instruction retirements and register updates. We do not focus on memory behavior, as it is

common for other golden model solutions, since that requires specialized solutions beyond the scope of this work. We then compress the collected information on-platform to minimize the amount of data transferred. As a result, we must only record and transfer a few bits per cycle, thus maintaining the acceleration performance advantage. The on-platform tracing logic is simulated along with the processor in the acceleration platform. To further minimize data recording, we do not track information that ties registers to a specific instruction; instead, we rely on the off-platform software, to reconstruct these connections based on the information recorded. Figure 5.6 presents an overview of our solution showing the components on the accelerator and on the off-platform software. It also outlines the type of data that is traced and transferred.

## 5.9 In depth view of the solution

This section presents an in depth view of the solution delineating the different aspects of it, namely: (i) which data is relevant for checking (ii) how data is compressed on-platform and (iii) how the off-platform software checker operates on the collected data.

### 5.9.1 On-platform data tracing

From a high level standpoint the collection of information for our purposes appears to be straightforward; however, when applied to an industry processor, many aspects become challenging. The processor in question is a modern, server class, superscalar out-of-order processor with simultaneous multi-threading allowing 8 simultaneous threads per core. Hence, each architectural event is a complex combination of several microarchitectural events. To correctly identify and log individual architectural events, we need a number of microarchitectural monitor points, mapped together with the design onto the accelerator. The main architectural events to be collected for our purposes can be grouped into the following 3 major classes:

**Instruction completion:** Since the underlying processor is out-of-order, we can only obtain a finalized instruction retirement event when an instruction is committed. This information is gathered from the group completion table of the processor design, where instruction completion events are built from micro-operation completion information.

**General purpose register activity:** This group of registers includes integer general purpose registers (GPR), floating point registers and vector registers (VR). Accessing update

events and values incurs an additional layer of indirection due to register renaming deployed in out-of-order microarchitectures.

**Special purpose register activity:** Special purpose registers (SPR), such as several status registers, are easier to handle, since they are directly mapped and have explicit signals that identify a write to a special purpose register. We chose to collect information on a subset of special purpose registers that are either part of or closely related to the architectural state.

Note that we record all instruction completion events and all update events on the monitored registers. However, we perform lossy compression on the data associated with each event, *i.e.* completed instruction addresses or values written to a register, to reduce the number of bits recorded on the acceleration platform.

### 5.9.2 On-platform data compression

As discussed in Section 5.8.2, a central goal of our work is to keep the amount of data recorded per cycle at a bare minimum, to maintain the performance advantage of acceleration, while still providing acceptable detection accuracy. To this end, we compress the data associated with each event, such as register update values and addresses of completed instructions. A lossy compression scheme, such as a checksum is ideal for this purpose, since we are only interested in identifying value deviations. So, as long as a different value produces a different checksum with high likelihood, it serves the purpose. Moreover, another important aspect in the development of our solution, is that the additional hardware required to implement the compression scheme should have minimal logic overhead and minimal logic depth. Hence, a compression scheme that involves little additional logic and does not add substantial delay to the critical path is favored over a more complex scheme.

#### Register update values

Value discrepancies in register updates can often be discerned using a checksum over a small subset of the bits, without requiring a complete value comparison. We strive to use only a few (say, less than 8) bits of encoded information for each register value field (32 bit / 64 bit). The basic idea is to compute a checksum from the value generated by the simulated hardware and perform the same operation on the value generated by the reference model for each register update in the software checker. For the sake of our checker solution, a checksum match is considered a valid register update. Since all checksum schemes are a hash function from a set of size $2^{64}$ (for 64 bit registers) to a set of size $2^c$, where c is a

small value, some amount of aliasing is unavoidable. However, we found that blocked parity schemes, presented below, provide sufficient accuracy in practice for the typical error scenarios that we encountered.

**Blocked parity schemes** partition the data vector into several distinct blocks and then compute single bit checksums for each block. The concatenation of these bits provides the final checksum. This approach is guaranteed to detect any bit value difference, as long as the number of single bit errors within each block is odd. A benefit of this approach is that its computation is extremely low cost in hardware, simply requiring a few XOR gates. However, this approach is ineffective for scenarios where errors manifest with an even number of localized bit-flips, which may occur all within one, or a few, blocks. To address this situation we build blocks on non-contiguous bits, scattering the bits over the checksum blocks. With this technique, an error affecting a few contiguous bits has a much higher chance of detection. The experimental evidence supports this intuition.

#### Retired instruction addresses

The data associated with each retired instruction is the address of the committed instruction. To compress these values we use a very simple scheme, recording only the last few bits of the address. Even though this scheme is prone to aliasing, it works very well in practice. Indeed, it allows us to identify an execution divergence from the golden model fairly precisely, since the probability of execution starting at an aliased address leading to the same sequence of register updates as the correct execution is extremely low.

#### Deciding checksum width:

We want to store a minimal number of bits in the checksum, while still detecting value discrepancies caused by a functional bug. Hence, we investigated the detection accuracies of several blocked parity schemes, as described in Section 5.9.2, over buggy traces diverging on a register value update.

To this end, we varied the number of checksum bits from 1 to 7, while the original register values are 64-bits wide. We studied three different checksum schemes as reported in Figure 5.7, and estimated the minimum checksum bit width required to detect typical value discrepancies. The schemes we evaluated are: (i) Simple blocked parity, where a single parity bit is computed from each portion of register data and appended to the final checksum. (ii) XOR sum of blocks, where the checksum is obtained by applying bitwise XOR to all same size sub-blocks of the register value. (iii) Overlapping block parity, sim-

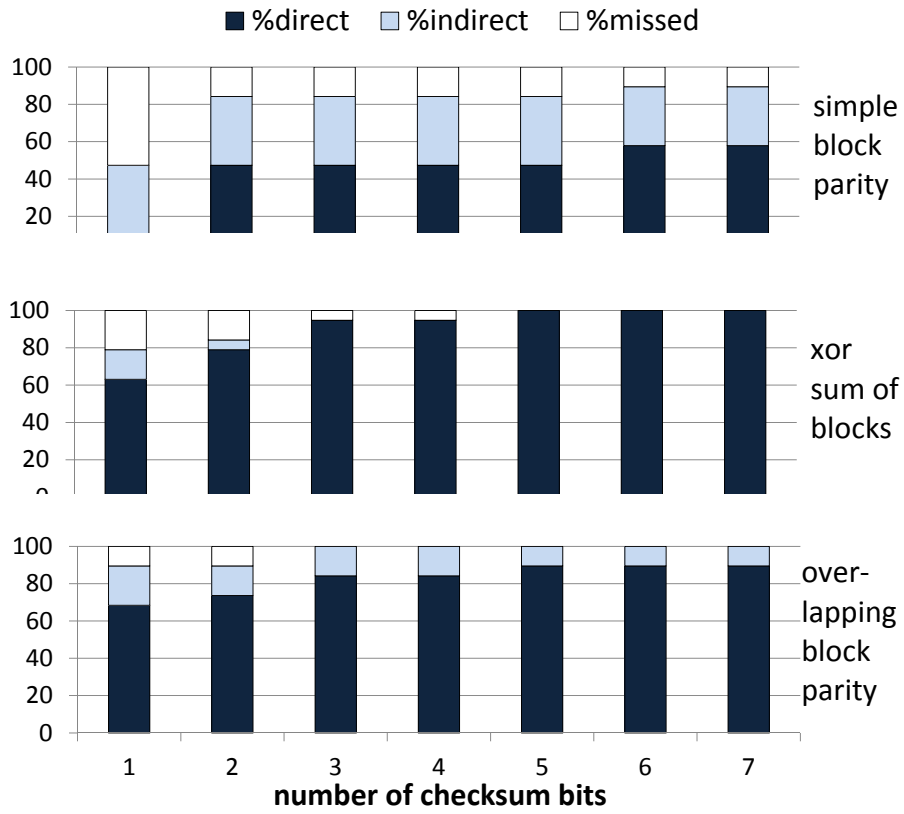



**Figure 5.7 Detection accuracy of a range of checksum schemes**. Register value discrepancies can be either detected at register update (direct), or in downstream computation (indirect), or missed.

ilar to (i), but with overlapping partitions. The sample size for this study was 500 traces with register value corruptions similar to those of actual buggy traces. From Figure 5.7, it can be gathered that typical discrepancies can be detected with as little as 5 bits of XOR sum of blocks.

### 5.9.3 Off-platform software checker

As discussed in previous sections, our instruction-by-instruction checker strives to identify all discrepancies between the simulated processor behavior and its golden model. A processor's architectural state is defined by the values of the architectural registers (including general purpose registers, certain special purpose registers that affect execution flow and program counter) and the contents of memory. We assume that events that are not captured by the golden model (such as memory updates due to shared memory) do not appear in the test case. Thus, our single core processor model can be considered to be executing correctly, as long as program flow and architectural state are identical to that of the golden model. Hence, tracking the completion of instructions (program flow) and any modification

to architectural registers is sufficient to check the correctness of execution. We encountered two key challenges in developing the off-platform checker, discussed below:

**Reconstruction of instruction flow:** A significant problem we had to address was the lack of close time correlation between an instruction retirement and its register events. This information cannot be reconstructed simply from the acceleration trace. Thus, in our solution we maintain a list of all registers that should have been modified by a completed instruction. We expect that for each such register, the first modification report that appears after the completed instruction will contain the correct value, and this report will appear within a bounded number of cycles. This solution is based on the assumption that registers are modified only after the corresponding instruction completes, and all associated register modifications are reported within a bounded number of cycles. However, we also had to consider the case where a register update is received before its corresponding instruction completion: in this case we must search for a matching event from the golden model over a few instructions downstream. If we do not find the matching event within a few instructions, we flag an error. We have run experiments to compare the results reported by a state-of-the-art software-based IBI checker to the results reported by our solution. We learned that the only difference lies in identifying which instruction is the root of the execution path deviation from the golden model execution (when such deviation exists). Our checker may report an instruction that is close to the actual deviating instruction (usually the next instruction), which we found satisfactory for effective debugging.

**Handling interrupts for checking purposes:** External interrupts and other non-deterministic events are not predictable by the golden architectural model; however, they are still included in the acceleration traces. External interrupts can still be identified from the address of the corresponding interrupt handler and specific values of the related control registers. Our solution mimics the effect of the interrupt routine by modifying the associated status registers and other architectural resources in the golden model and then it resynchronizes the model with the trace.

## 5.10 On-platform tracing unit

As discussed earlier, there are several types of data collected on the acceleration platform originating in different regions of the design at a variable rate. To manage this flow of data, we developed a novel unified scheme to collect and organize it for on-platform storage, before it can be transferred off-platform. To this end, we first need a mechanism to identify which registers are updated on a particular cycle or which instruction groups have

completed, so that we only record new values for the relevant registers/addresses. Second, we need a mechanism to present this data in a structured fashion, so that it can be recorded efficiently by the acceleration platform's data logging mechanism. We note that, although the maximum number of simultaneous events in a clock cycle can be quite high, the average number of events per cycle is fairly small. Hence, a recording mechanism that can handle transient peaks in the number of events and can present data at a constant rate to the platform's debug support unit would be ideal. A possible solution to this second requirement is a first-in first-out buffer that allows the storing of up to a few entries at a time and it is drained at a constant rate. This section discusses how we achieved these requirements.

### 5.10.1 Select and encode logic

The first task of the tracing unit focuses on selecting and encoding different types of events as they are flagged during a clock cycle. In the platfom there are a number of data lines and corresponding valid lines coming from different parts of the processor and corresponding to different special purpose registers or instruction completion events that we want to track. Our goal is to be able to store the relevant data at each cycle (as signaled by the corresponding valid lines) while also tracking the correct source for the data. By doing so the off-platform software is able to reconstruct the sequence of events to be checked against the golden model.

The goal of the select and encode logic unit can be formally expressed as follows: given a collection of $N$ signal lines, presented as an ordered list, up to any $M$ lines among those can request data logging on any given clock cycle. The task of this unit is to identify and encode the position in the list of the $M$ lines in preparation for storing them along with the data itself. Ultimately, these positions will be used to identify the source of the corresponding data value. This problem is also known as the "detect and encode all ones" problem: one straightforward solution would be to use a chain of priority encoders: the first encoder is responsible for the highest order position, which is then masked and the entire vector of N lines is passed down to the next encoder. While simple, this solution creates a deep combinational logic block, which could hamper the performance of acceleration.

Our goal in developing this unit is to develop a design that is most suited for acceleration platforms, even if it may entail a non-minimal area footprint in silicon. To this end, we devised an alternative solution, that has a much smaller logic depth. Our solution uses a parallel detection scheme, where each detection block is responsible for generating a one-hot encoded vector corresponding to the line position for which the block is responsible, if that line has data available. If no logging data is generated from that line during a cycle,

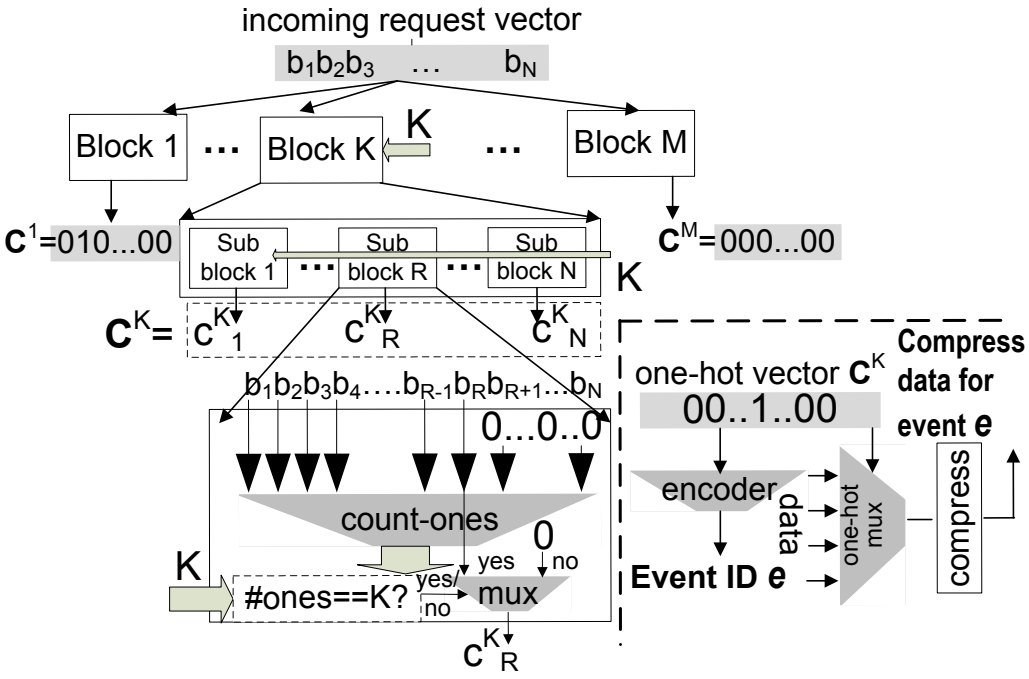



**Figure 5.8 Detector block to identify the source of data** to be logged in a given clock cycle of simulation acceleration.

the block should simply output a vector of zeros. Figure 5.8 illustrates our solution: we use $M$ detection blocks, since we have at most $M$ lines generating data within one cycle. Each block receives in input a value $K$, and generates a one-hot encoded vector where the 1-bit is in the position of the $K$-th line producing data in that cycle. For instance, if during a cycle lines 4, 7 and 11 produce data to be logged, then block 1 should have a one in position 4, block 2 should have a one in position 7 and block 3 should have a one in position 11.

### 5.10.2 Trace buffer

Once the relevant data has been selected and encoded for logging, we need a hardware block to record the architectural events. To this end we use a trace buffer that must be capable of handling up to $M$ entries in each clock cycle, while allowing a constant $R$ entries to be read. Such buffers are typically realized via a circular buffer with read and write pointers. However, multiplexors are needed to realize these pointers. Unfortunately, they also increase the logic depth of the design, particularly when the number of buffer entries is large. Hence, we adopted an alternative design, where the buffer is implemented as a shift-buffer, so that the constant number of read operations in each simulation cycle corresponds to a constant number of shifts. A bit is associated with each entry to indicate the first free entry, and independent write units are associated with each buffer entry. Each write unit

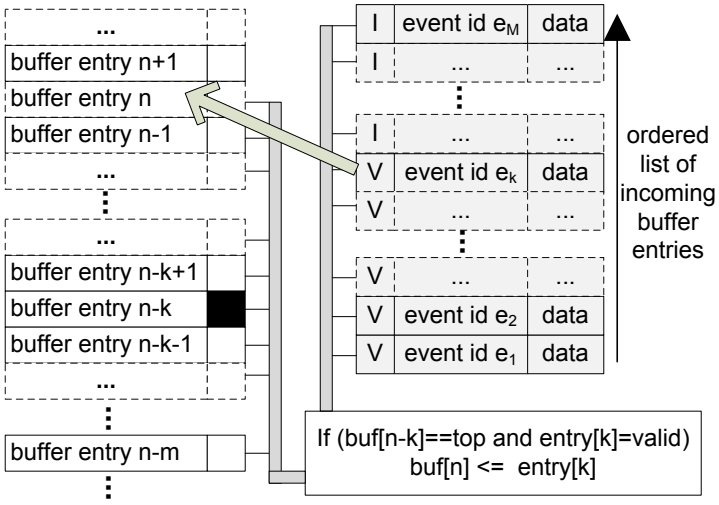



**Figure 5.9 Trace buffer writing unit**. Each buffer entry is associated with a writing unit. Each unit determines which data logged in the cycle should be stored in the position for which it is responsible.

has access to its corresponding entry and the $M$ preceding ones, and it determines what to write in its entry based on the number of write operations to be completed in the cycle. This design is shown in Figure 5.9: the implementation is parallel and logic depth is kept minimal. We derived a queuing theory-based estimate for our buffer size, which ensures a very low probability of overflow, while using the lowest possible draining rate.

## 5.11 Experimental evaluation of the IBI solution

Our solution was implemented for an upcoming POWER processor core design on the AWAN accelerator [30] platform. We evaluated the capability of our solution to detect bugs as well as its performance. The IBM SixthSense tool-chain was used to design and synthesize the hardware blocks for our solution. The processor core netlist consisted of a few million logic gates, and the resulting logic overhead was within 20%.

### 5.11.1 Bug detection capability

Any discrepancy of the processor's behavior from the golden architectural model due to a probable functional bug is detected as one of the following situations (symptoms) by our IBI checker:

1. **Register value mismatch:** Updated value of a register does not match with predicted

value from golden model;

**2. Unexpected register update:** An architectural register update event takes place in the design but not in the golden model;

**3. Unaccounted register update:** A register update event takes place in the golden model but does not occur in the design;

**4. Wrong instruction:** The instruction address of an executed instruction is in disagreement with the golden model;

We obtained a set of 145 architectural event traces that exposed actual functional bugs. These 145 constituted the entire set of buggy traces that we had access to. To evaluate the bug detection capability of our checker, we ran the same traces on our off-platform software checker to determine if our accelerator-based checker could also detect the occurrence of the bugs. All 145 testcases exposed a bug in our setup; in addition the symptoms reported matched those of the software-based golden model solution. We report in Table 5.4 the distribution of the bugs detected according to the type of symptom flagged by our checker. As it can be noted, a large portion of the issues are due to unaccounted/unexpected register updates. All these problems were detected within 5 instructions from the first point of golden model/accelerator divergence.

| Symptom | #occurences |
|---|---:|
| Register value mismatch | 21 |
| Unexpected register update | 30 |
| Unaccounted register update | 89 |
| Wrong instruction | 5 |

**Table 5.4 Distribution of bugs detected by our solution**.

Since we do not compress the information regarding which architectural register (among the monitored subset) is updated, we detect all discrepancies that are not affected by checksum aliasing. However, even in this latter case, often the program flow diverges substantially due to the bug, and we can still flag the issue a few instructions downstream.

### 5.11.2 Tracing overhead

The amount of logic added for on-platform tracing purposes may impact the performance of the simulation. However, this is only the case if the overall logic size mapped to the platform (design + checkers) exceeds a certain threshold, dependent on the accelerator's characteristics. When abiding this threshold, the performance degradation due to the tracing logic comes from two sources (i) additional logic to simulate (ii) signal recording time.

To evaluate these effects, we measured the simulation acceleration performance of the

POWER core design in several situations. The stimuli used for this study were regression tests lasting few million cycles. First, we run a baseline design with no tracing logic. Then we added the tracing logic, but without observing the trace buffer output. Then we also enabled tracing for the typical case, that is, 3 buffer entries are read per cycle, amounting to 50 bits of recorded information per cycle. Finally, we considered an extreme situation where 10 buffer entries are read per cycle, for a total of 162 bits. Figure 5.10 summarizes our findings, normalized to the simulation performance (between 10-100 kHz) of the baseline design with no tracing logic.

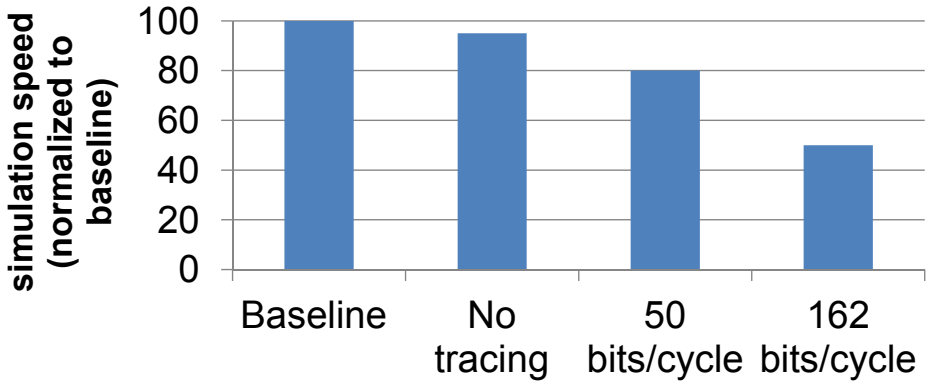



**Figure 5.10 Impact of tracing logic** on acceleration performance.

From Figure 5.10 we gather that our solution introduces only a 5% slowdown due to the tracing logic alone, and another 15% due to data logging. Even the extreme situation causes no more than a 50% slowdown in acceleration performance, a value still order of magnitudes better than software-based simulation.

## 5.12 Related work

A rich body of solutions is available for the validation of high-level behavioral models of digital designs, both spanning constrained test generation and formal property verification, enabling designers to specify complex assertions/checkers and expose bugs. Correspondingly, a wide range of languages exists to describe the structure and concepts needed: *e*, Vera, SystemVerilog, C++, *etc*. Unfortunately such rich environment does not carry over to hardware platforms for validation, such as simulation acceleration, emulation, or silicon debug. The main focus of several research works in the past decade has been the efficient synthesis of formal assertions into realizable hardware description [2, 31]. These techniques target specifically acceleration, emulation or in-silicon debug [18, 20]. Reconfigurable designs for debug architectures, enabling verification engineers to create assertion

checkers, transaction identifiers, triggers, and event counters in silicon have also been suggested [3]. However, assertion synthesis is an exact translation of individual properties and can generate extremely complex logic blocks, which can reduce or eliminate the acceleration advantage. In this work we focus on limiting the logic overhead of the embedded checkers by leveraging approximation.

The possibility of adopting conventional software testbenches for acceleration and emulation platforms has been considered in prior work as well. The testbench still executes in software and communicates with the platform over a bus: in this setup the communication often becomes the bottleneck [67, 56]. Transaction-based acceleration (TBA) [85] attempts to overcome this bottleneck by bundling several interactions between the testbench and the platform into larger, yet less frequent transactions.

Approximation of logic functions has been proposed in other related domains. For example, it has been applied as a method of restricting the size of a binary decision diagram(BDD)[82]. Our objective with this work is to reduce the size of the circuit representation of a Boolean function. The techniques applied in timing speculation [38], and typical-case optimization [9] can also be viewed as logic approximation, since the active circuit during a given clock cycle is an approximation of the complete circuit function.

Simulation accelerators and emulation platforms have been traditionally used to boost the productivity of the microprocessor validation effort [43, 79], and they play an even more critical role today, in light of the increased complexity of these designs. Existing acceleration-based flows usually have a coarse checking granularity, that is, they can label a test as passed or failed after its completion but, in case of failure, no additional information is available related to the time/location of the bug manifestation. Comparing architectural state between a purely software-simulated design model and a golden architectural software model at instruction boundaries, or at other synchronizing boundaries, has also been a commonly deployed method for microprocessor validation[97, 26]. The key reason why this methodology was not considered for acceleration, with the golden model running in software on a host platform, is that connecting these two components (golden model and accelerated design) is both difficult (due to lack of debugging support) and detrimental to performance [26]. Obtaining scan values from a silicon prototype and comparing them to a RTL golden model to detect divergence analysis during post-silicon debug has been proposed in [24]; however, this solution is only used to diagnose electrical faults.

More recent silicon-debug solutions, such as IFRA [76], introduce additional logic into the design to trace the flow of an instruction through various microarchitectural blocks and use this information with a post-simulation analysis tool to locate the manifestation of a possible design bug. Though our solution has a similar organization, *i.e.*, decoupled trac-

ing and checking components, we are interested in the manifestation of a failure in the architectural state. Moreover, IFRA cannot detect divergence of the processor execution from the ideal model on its own, because it solely relies on post-triggers for this information. Our solution is focused on detecting the first point of divergence in the architectural state, hence it solves an orthogonal problem. Certain runtime verification techniques such as DIVA [8], introduce a lightweight companion processor to check the architectural state of the main processor, but these solutions operate at runtime, past design debug.

## 5.13 Summary

In this chapter two key solutions to bring in checking capabilities into acceleration platforms were presented. These solutions enable hardware-accelerated simulation platforms to become fully effective towards performing simulation-based validation. The first solution, namely checker-approximation enables efficient mapping of software checkers to hardware-accelerated simulation platforms. The second solution provides a novel scheme of leveraging on-platform logic to perform compression to reduce the amount of data to be transferred off-platform for checking.

Approximation trades off logic complexity with bug detection accuracy by leveraging novel techniques to approximate software checkers into small synthesizable hardware blocks, which can be simulated along with the design on a hardware-accelerated simulation platform. I presented a generalized checker taxonomy, proposed a range of approximation techniques based on a checker's characteristic and provided metrics for evaluating its bug detection capabilities. The case studies have demonstrated that checker approximation is a viable solution to reduce hardware overhead of complex checkers while still enabling detection of a large fraction of bug manifestations.

The second solution was demonstrated with a novel microprocessor design checking scheme that provides architectural checking against a golden model for simulation acceleration. On-platform logic was used to trace and compress necessary data for checking, reducing the number of signals to be traced, which is key to retaining the performance advantage of hardware-accelerated platforms. The solution provides the same bug detection quality as its software-based counterpart. It enables architectural validation of the design on acceleration platforms with negligible accuracy loss. Thus it makes microprocessor design validation possible at an order-of-magnitude better simulation performance than software-based simulation.

We discussed two separate solutions to bring in checking capabilities beyond software-

based simulation in this chapter. The next chapter will explore how these two solutions can be leveraged in a combined fashion, each complementing the other.

# Chapter 6
# Hybrid Checking

The previous chapter explored two approaches to introduce checking capabilities into hardware-accelerated platforms. Checker approximation brings in such capabilities in form of low overhead logic, which is simulated alongside the design. In contrast, the IBI solution leverages additional logic to compress simulation logs necessary for checking, while the actual checking activity is performed off-line on the compressed log. Case studies for both solutions were performed on microprocessor designs. We note that both solutions are incapable of harnessing simulation acceleration to the fullest extent due to their inherent limitations. Certain full-fledged software-based checkers with complex functionality do not have efficient hardware representation. Checker approximation cannot incorporate those checkers without heavily sacrificing accuracy. On the other hand, if all the necessary checks for a design block are performed offline, then the volume of recorded data may become large enough to erode away the performance advantage of the platform. Hence, although these solutions are very successful for specific kind of checkers, neither is able to provide the same degree of checking capability as software-based simulation while maintaining the platform-specific performance advantage in a general setting, when applied individually. It is interesting to note that a synergistic application of the two key ideas behind these solutions is likely to be more successful. This chapter investigates this possibility.

This chapter introduces a solution called "*hybrid checking*" which synergistically leverages both embedded logic and post-simulation software checkers to provide high quality checking capabilities with tolerable performance overhead. The key idea involves intelligently dividing checking responsibility between embedded checkers and post-processing software, such that certain aspects of checking are performed by embedded checking logic, while certain other aspects that must adopt the "log and then check" approach are performed off-line with software checkers. First of all, Embedded checking logic reduces the amount of information that needs to be checked off-line. Then, on-platform compression logic is leveraged to further reduce log size. This methodology is demonstrated by adapting

some of the checkers in the software-based verification environment for a modern microprocessor design for acceleration platforms in a preliminary study. Even though this case study targets acceleration platforms, the solution can be applied to other types of hardware-accelerated simulation platforms as well. We provide insights on how the checking activity for a component design block can be partitioned into the two aforementioned categories. Finally, we demonstrate novel techniques to reduce the amount of recorded data with the aid of lightweight supporting logic units, leading to only a marginal checker accuracy loss. These techniques, when applied in conjunction, are able to realize the final objective of this dissertation: performing efficient validation with high-performance simulation.

## 6.1 Towards hybrid checking

As discussed in Chapter 2 software-based simulation provides a feature-rich environment for verification, which is critical in validating and debugging a design. A number of checkers are connected and simulated with the processor design at various phases. The tight coupling between the checker functions and simulated design allows for a relatively low effort checker design. Often, these checkers include end-to-end correctness checks for correct architectural execution and memory access protocols, as well as localized checkers for individual microarchitectural blocks. **Checker-centric validation**, although very successful for software-based simulation, **does not extend to acceleration or emulation environments in a straightforward manner**. As pointed out in Chapter 2, hardware-accelerated simulation platforms can only simulate synthesizable logic; hence, even though the design can be synthesized and simulated at high performance, the testbench and checking environments do not extend into the realm of hardware-accelerated platforms [57]. Moreover, **lockstep execution of software checkers on a host paired with the design simulated on a hardware-accelerated platform is not tenable**, since it degrades overall performance to an unacceptable level.

It is, therefore, critical to adapt checkers to these platforms to fully leverage high-performance simulation for validation and debugging. Current methodologies on this front have focused on limiting the number of synchronization events between the host running the checkers and the accelerator by: i) accumulating short and frequent interactions between the design and the testbench into longer and infrequent transactions [57, 85], ii) recording the values of critical design signals during simulation on-platform, and then off-loading the log at the end to check for consistency with a software checker [27], iii) synthesizing some of the checkers into hardware for simulation alongside the design [19, 63].

None of these approaches provide a definitive and complete solution as they suffer from simulation **slowdown due to large log transfers or large logic overhead**. The proposed solution strives to overcome both of these shortcomings.

Any checking solution adapted for hardware-accelerated platforms must consider several trade-offs regarding checking capability and performance. As discussed in Chapter 5, recording a large number of signals during simulation can incur a performance penalty due to inherent constraints of the acceleration platforms. Thus, in such an approach, the average number of recorded bits per simulation cycle must be small enough to introduce only an acceptable degree of slowdown. Adding extra logic to be simulated on the platform alongside the design, if there is room, can also cause slowdowns. Hence, if embedded logic (such as synthesized checkers) is to be simulated with the design for checking purposes, it must be as small as possible. In an effort to reduce recorded bits and synthesized checker logic, some of the capabilities of the original software-based checkers may also be lost. In view of these constraints, **a desirable solution towards checking on acceleration platforms should have minimal logic footprint and record only a small number of bits per cycle, while providing the same quality of results as the original checker in software-based simulation**.

### 6.1.1 Overview of this chapter

In this chapter, we propose a novel checker adaptation methodology (see Figure 6.1) for microarchitectural blocks, which synergistically uses embedded logic and post-simulation software checkers to provide high quality checking capability with very small performance overheads. **Checkers that have a small logic footprint when synthesized can be embedded and simulated** with the design – we call these "**local assertion checkers**". On the other hand, **checkers that must adopt the "log and then check" approach because of their complexity**, compress activity logs relevant to the check using **on-platform compression logic and then perform the check off-platform** – we call these "**functionality checkers**". This concept is explained in-depth in Section 6.2.

We demonstrate our methodology on the software verification environment for a modern out-of-order superscalar microprocessor design. The simulation-based verification environment for the processor is equipped with architectural as well as a number of microarchitectural block checkers designed for software-based simulation. Adapting these checkers to an acceleration environment provides a challenge that is representative of those faced by verification engineers working in the field. To give an example, if a checker responsible for a certain microarchitectural block is adapted for acceleration naively by a "log

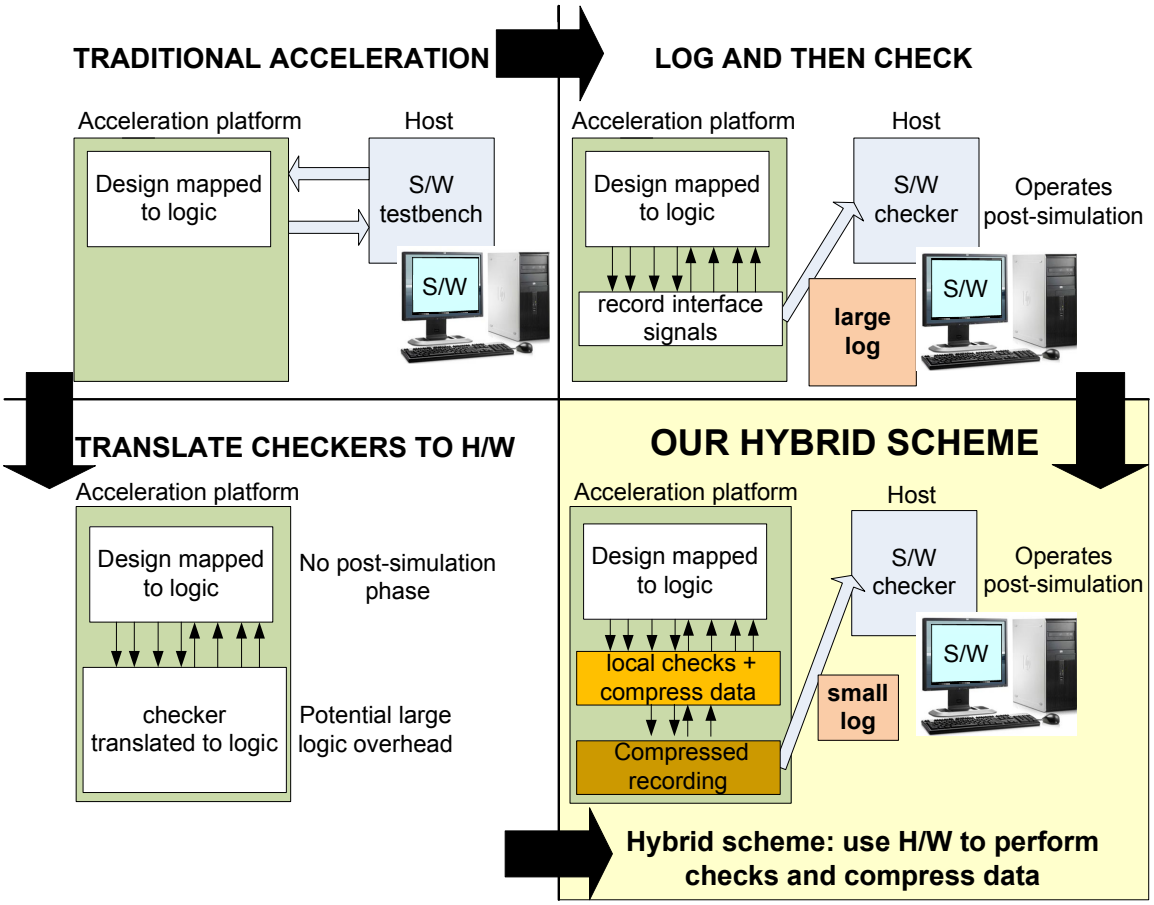



**Figure 6.1 Hybrid checker-mapping approach**. We use a mix of embedded logic and data-compression techniques for data that must be logged. For the latter, a software checker analyzes the logged data after simulation.

and then check" approach then it will necessitate recording all the input and output signals of the block. Such excessive recording would introduce unacceptable degree of slowdown and defeat the advantage of acceleration. Moreover, for quality debugging we need to accommodate as many microarchitectural checkers as possible, such naive adaptation would only allow incorporating a small number of them. Translating all microarchitectural checkers into synthesizable logic can also be untenable due to their complexity or the introduced logic overhead.

We provide insights on how the checking activity for a microarchitectural block can be **partitioned into local assertion checkers and functionality checkers** in Section 6.2.1. Following Section 6.3 explores **novel techniques to reduce the amount of recorded data for functionality checking** with the aid of lightweight supporting logic units. The particular design used as a testbed for hybrid checking is introduced in Section 6.4. The various trade-offs introduced by our solution are demonstrated in Section 6.5 for a single representative microarchitectural block. Relevant prior work is presented in Section 6.6 and finally

Section 6.7 concludes this chapter.

## 6.2 Synergistic checking approach

The most common method of checking microarchitectural blocks involves implementing a software reference model for the block. The design block updates a scoreboard during simulation, which, in turn, is checked by the software reference model [94]. This approach is viable in software simulation, but not directly applicable to acceleration platforms. Since acceleration platforms only allow simulation of synthesizable logic, one option is to implement the reference model in hardware; however, this option is often impractical. Another option is to record all signal activity at the microarchitectural blocks' I/O and cross-validate it against a reference model maintained in software after the simulation completes. However, that solution requires recording of a large number of bits in each cycle, leading to an unacceptable slowdown during simulation. Thus, neither solution outlined scales well to complex microarchitectural blocks. We propose a two-phase approach that solves this problem by making synergistic use of these two methods while avoiding the unacceptable overheads of both.

The first phase performs cycle-by-cycle checking using embedded local assertion checkers on-platform. It focuses on monitoring the correctness of the target block's interface activity and local invariants, which can be expressed as local assertions. During this phase, we also log and compress (with embedded logic) relevant microarchitectural events to enable off-platform overall functionality checking. In the second phase, the logged data is transferred off-platform and compared against a software model to validate the functional activity of the block. This approach is illustrated in Figure 6.2. The main idea behind this two-phase approach is the separation of local assertion checking from functionality checking for a design block.

**Local assertion checking** often requires simple but frequent monitoring. Hence, it must be performed in a cycle-accurate fashion and can often be achieved via ***low overhead embedded logic***, with minimal platform performance loss. This is because most local assertions are specified over a handful of local signals and can be validated in an analysis' windows of a few cycles (*e.g.*, a FIFO queue must flush all of its content upon receiving a flush signal, FIFO head and tail pointers should never cross over, *etc.*). These checkers do not require large storage of intermediate events, rather they must maintain just a few internal states to track the sequential behavior of relevant signals.

In contrast, **functionality checking** can be carried out in an event-accurate fashion.

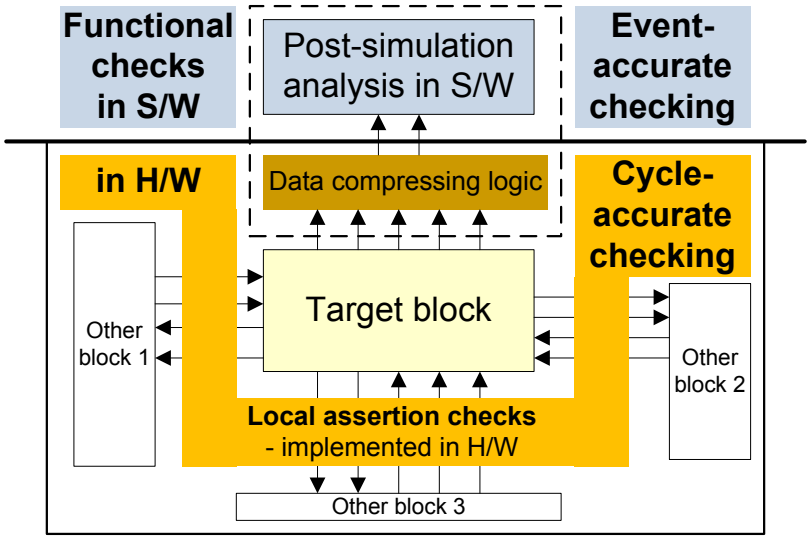



**Figure 6.2 Two-phase checking**. Local assertion checks are performed by embedded logic in a cycle-accurate fashion, while microarchitectural events are logged and compressed on platform with additional logic, and then evaluated for correctness by an off-platform functionality checker after simulation.

From a functionality perspective, most microarchitectural blocks can be abstracted as data structures accessed and modified through events of read and update operations. The main goal of functionality checking is then to verify the legality and consistence of operations on this data structure. In addition to monitoring the data associated with events, an event-accurate checker also needs to perform bookkeeping of the internal contents of the microarchitectural block, and thus an embedded logic implementation would be grossly inefficient. Therefore, for functionality checking, the data associated with events should be recorded and transferred off-platform for ***post-simulation analysis in software***, where the validity of the recorded sequence of events is checked. Since events need to be recorded only as they occur, there is no need to log signal values on every simulation cycle. Moreover, we notice that we can further reduce the amount of data recorded by leveraging ***on-platform compression***, while still achieving high-quality functionality checking.

### 6.2.1 Checker partitioning

It is technically possible, though inefficient, to express any collection of checks entirely as an embedded hardware or entirely as a post-simulation software checker (preserving cycle-accurateness via tracing cycle numbers, if needed). The partitioning of software-based checkers into local assertions and functionality checkers requires the involvement of a verification engineer who can extract the aspects that can be mapped into local asser-

tions. However, there are high-level guidelines that we gained from experience and that can be used to guide and simplify this task. As discussed above, verifying the high-level functionality of a block is naturally a perfect fit for event-accurate functionality checking, whereas verifying simple interface behavior and component-specific invariants with cycle bounds is a better fit for local assertion checking. The primary criterion when making this distinction should be whether event-accuracy is sufficient or cycle-accuracy is needed to implement a check. Another governing principle is that the logic footprint of a synthesized local assertion should be small. Hence, a sufficiently complex interface check that will result in a large logic overhead upon synthesis should be implemented as a post-simulation software checker instead. Once a checker is selected for local assertion checking, it can be coded as a temporal logic assertion and synthesized with tools such as those in [2, 20]. Note, however, that for our initial experimental evaluation, we simply coded the assertions directly in synthesizable RTL.

## 6.3 Functionality checking with on-platform compression

We have observed that most microarchitectural blocks can be evaluated as black-boxes that receive control and data inputs from other microarchitectural blocks or shared buses, and either output information after some amount of processing, or perform internal bookkeeping. From a checking perspective, microarchitectural blocks can essentially be treated as data-structures, where data-elements are allocated, updated and deleted on external trigger events. The objective of functionality checking is to ensure that the log of events is consistent as per the rules of the operation of the microarchitectural block. The role of on-platform compression is to compress the data associated with the events in a fashion that allows us to perform checking, while reducing the volume of the data to be transferred off-platform.

In our solution, functionality checkers gather all the relevant events for each microarchitectural block by logging the associated control and data signals on-platform for later transfer and post-simulation analysis. During the logging process, however, we also compress the collected event log so as to reduce transfer time. Our goal is to achieve compression in the recorded information without sacrificing accuracy. From a verification perspective, control signals are more informative than data signals; hence, the guiding principle is to preferentially compress data content over control information. Indeed, the control information is critical in keeping the post-simulation software checker in sync with the design block. Since compression is performed using an embedded logic implementation, we want

to leverage low-overhead compression schemes, such as parity checksums, which can be computed with just a few XOR gates.

In Chapter 5 it was shown that blocked parity checksums are generally sufficient to detect value corruptions due to functional bugs in modern complex processor designs. In light of this, a straightforward technique consists of **compressing the data portion of all events using a blocked parity checksum, while keeping control information intact**. Thus post-simulation software checker is able to follow the same sequence of control states as the design block to validate its behavior. Moreover, some types of events may undergo additional compression steps as discussed below. Taking advantage of the relative importance of control and data signals further, it is sometimes sufficient to record all events with their corresponding control signals, and simply drop the data components of the event. Thus the post-simulation software-checker can follow the **same sequence of control states while the associated data is only sampled on a subset of cycles**. Another technique consists of **merging checksums across multiple events**. Instead of checking individual output events, we can construct their "checksum digest" spanning multiple events and validate this digest against the golden model.

## 6.4 Case-study design

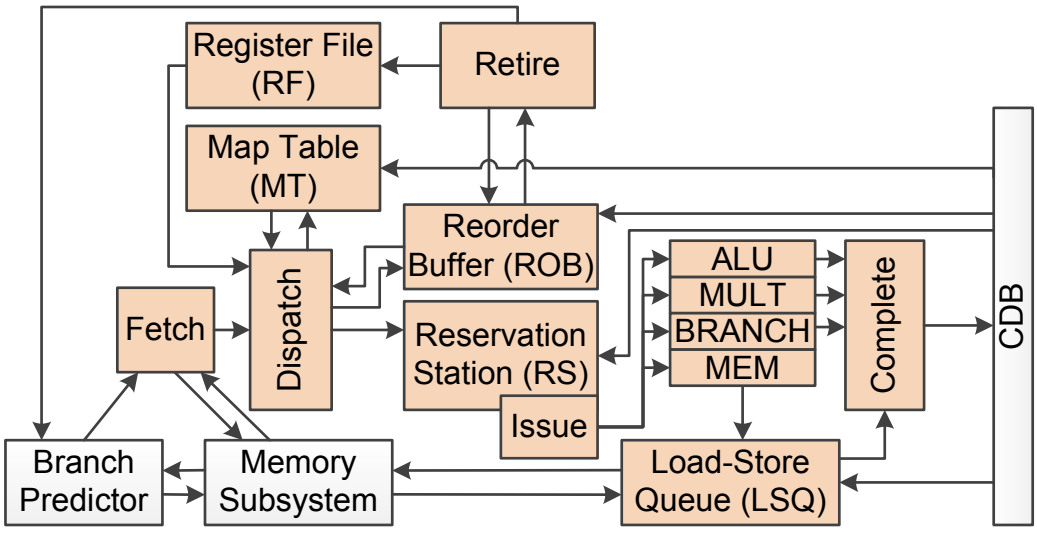



**Figure 6.3 Microarchitectural blocks** in our experimental testbed.

A 2-way superscalar RISC out-of-order processor core designed for a subset of the alpha ISA serves as the case-study for this work. The microarchitecture of this design is based upon the intel P6 microarchitecture. The main microarchitectural blocks that hold

state information in the core are reservation station (RS), map table (MT), reorder buffer (ROB), functional units (FU), load-store queue (LSQ) and register file (RF). These along with the fetch unit, decode unit, dispatch unit, issue unit and a common data bus (CDB) aided with a bus arbiter forms the out-of-order core. In this microarchitecture each dispatched instruction is tagged with its corresponding ROB id during its lifetime of execution. All dispatched instructions are allocated on the RS, and wait there to be issued till both their source operands are ready, an entry for the instruction is also created in ROB and and possibly in LSQ, if its a memory operation. Once issued, the instruction is processed in the appropriate functional unit and when completed the computed destination register value along with the instruction's tag is put on the CDB. The ROB entries update based on the completion information on the CDB, and retire instructions in-order.

Our verification environment consisted of multiple C/C++ microarchitectural block-level checkers, one for each of the blocks reported in Figure 6.3, and an architectural golden model checker (*arch-check*) connected to the design via a SystemVerilog testbench. The block-level checkers implement behavioral golden models for each such block and check whether the output events of the block are consistent with the sequence of input events. We also equipped our verification environment with a time-out condition on instruction retirement, indicating whether the processor had hung ($\mu P$ *hang*).

Developing acceleration-based checkers for state-heavy blocks such as RS, ROB, LSQ *etc.* has been traditionally a challenge as: i) if the checker is entirely implemented in hardware, the logic overhead becomes unacceptable – comparable in size to their design counterpart, and ii) these blocks generate many events, thus logging entails lots of storage, data transfer and analysis. Hence, we believe that the validation of these blocks will benefit the most from our solution.

## 6.5 Experimental evaluation of hybrid checking

We performed a preliminary study on the feasibility of our hybrid checking methodology by analyzing several schemes on the checkers in our testbed. The validation stimulus was generated using a constrained-random generator that created a test suite of assembly regressions. In evaluating the quality of our solution, we considered the three most relevant metrics: **average number of bits recorded per cycle**, **logic overhead** and **checking accuracy**. The first metric reflects the amount of data to be recorded on platform and later transferred; we estimated the second one by using tracing logic similar to our IBI checking solution described in Chapter 5; the third one is obtained by comparing our hybrid checkers

against the bug detection quality of a software-only checker in a simulation solution. Note that, our industry experience suggests that the average bits/cycle metric is the most critical for acceleration performance. A recording rate of only 162 bits/cycle is reported to induce a 50% slowdown for the acceleration platform used in [27].

We injected a number of functional bugs in each microarchitectural block to evaluate the bug-detection qualities of our solution. To measure the checking accuracy of any compression scheme, the full set of regressions were run with only one bug activated at a time, and this process was repeated for each bug to create an aggregate checking accuracy measure. Each microarchitectural checker was only evaluated over the bugs inserted into its corresponding design block. We present preliminary results of applying hybrid checking on a single microarchitectural block namely, an arithmetic logic unit (ALU): a functional unit block.

### 6.5.1 ALU Checker

All functional unit checkers can be easily adapted into our hybrid checking methodology. Local assertions for these blocks include checkers validating the time of the completion of an operation and the corresponding release of the results to the ROB. Functionality checking requires logging operands and corresponding results for validation by the off-platform architectural checker. In this section we present the results relating to the ALU checker. We modeled five distinct functional bugs (see Table 6.1) on the ALU block to evaluate the quality of our checking schemes.

| ALU's functional bugs | |
|---|---|
| - Erroneous handling of the immediate field | - Erroneous logical operation |
| - Erroneous arithmetic operation | - Erroneous comparison operation |
| - Erroneous interaction with CDB | |

**Table 6.1 ALU checker - injected functional bugs**

The ALU receives instructions with operand values obtained from the issue buses, and outputs the result on the CDB after computation. Associated with each instruction are data signals: two 64-bit operand values on the input side, one 64-bit result on the output side and a 6 bit-wide control signal (the tag), on both directions. For this block, the **only local assertion** checked **whether an ALU instruction completes within 1 cycle excluding stall cycles**, while **functionality checking was used to verify computation performed by the block**. Recording data checksum for each ALU instruction while following the complete sequence of instructions is a natural choice for compressing events for the functionality checker; using a checksum scheme on the data output while preserving the whole 6 bits

of control. Truncation can also serve as a checksum scheme, though it is generally less effective than a XOR checksum for the same bit-width. However, to verify the result of the computation, we still need all operands' bits for each passing instruction. We can also merge successive results in one digest and record the digest, still needing all the operand bits. Finally, sampling can also be used as long as we keep track of all instructions going through the block but record operands and results in a sampled fashion. Table 6.2 details the set of compression schemes that we used in evaluating this checker.

| Name | Compression scheme |
|---|---|
| **csX** | compress 64-bit output into X parity checksum bits |
| **trunc** | check only 8 least significant bits of output |
| **merge** | merge results of 5 consecutive instructions going through the block |
| **samp** | record data for only 1 out of 5 instructions going through the block |

**Table 6.2 ALU checker - Compression schemes**

Figure 6.4 explores the trade-off between the checking accuracy of different data compression schemes and their average recording rate: the first bar on the left is for the the full software checker tracing all active signals and leading to an average rate of 25 bits/cycle. In contrast, the hardware-only checker does not entail any logging. Note that the average recording rate is much smaller than the total number of interface signals for the block, since only a fraction of instructions require an ALU. The other bars represent truncation, sampling, merging (low logging rate and low accuracy), and various checksum widths. Note that our checksum compression scheme provide very high accuracy at minimal logging cost. We believe this is due to i) the ability of checksums to detect most data errors, and ii) the fact that some control flow bugs impact data correctness as well. Logic overhead corresponding to these schemes are reported in Figure 6.5.

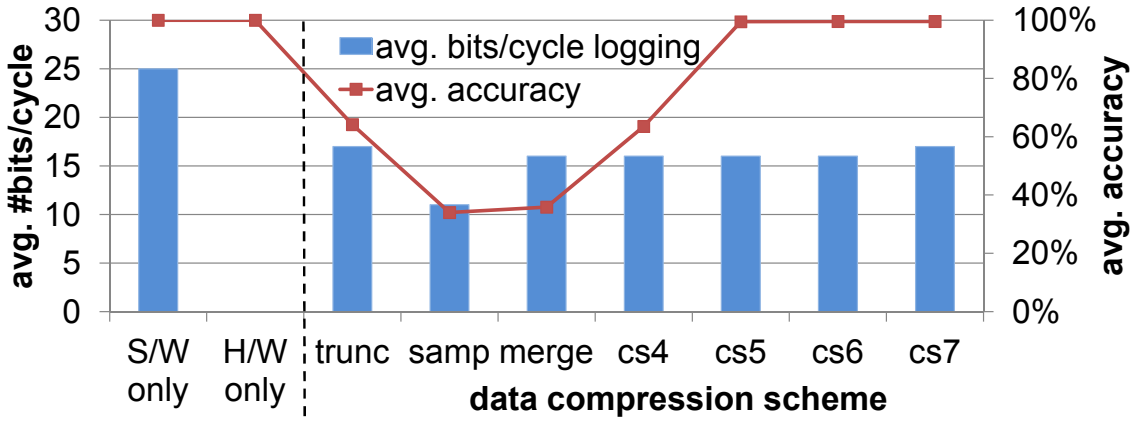



**Figure 6.4 ALU-checker - Accuracy vs. compression**. Note how the ALU-checker exhibits perfect accuracy even with a low checksum width of 5 bits.

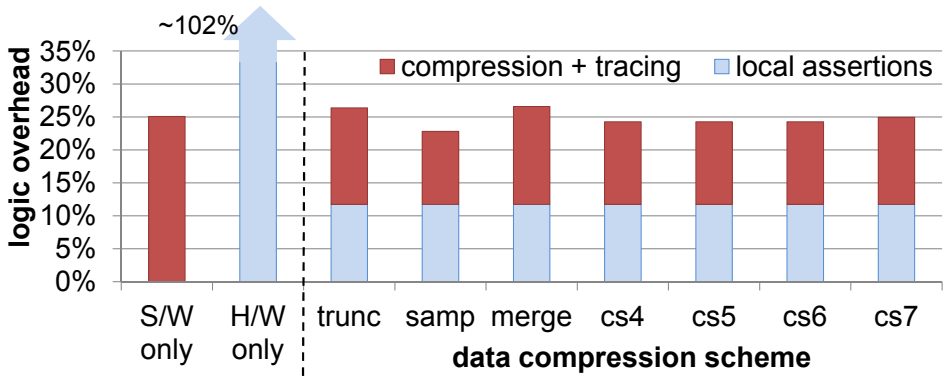



**Figure 6.5 ALU-checker - Logic overhead** relative to the ALU hardware unit for a range of compression schemes.

## 6.6 Related work

A plethora of solutions are available for simulation-based validation of digital designs using software-based simulation [94]. A simulation-based validation environment commonly involves checkers that are connected to the design. These checkers are written in high-level languages, such as C/C++, SystemVerilog, and interface with the design via a testbench. Unfortunately such validation schemes cannot leverage the performance offered by hardware platforms for validation, namely simulation acceleration, emulation, or silicon debug. Prior research has investigated synthesis of formal temporal logic assertions into synthesizable logic [2, 31], targeting those platforms [18, 20]. Techniques for using reconfigurable structures for assertion checkers, transaction identifiers, triggers and event counters in silicon have also been explored [3]. However, synthesizing all checkers to logic is often not viable for multiple reasons. Software checkers are often developed at a higher level of abstraction for a design block, thus a direct manual translation to logic will run into the challenge of addressing logic implementation details and can be error prone. Though these checkers can be translated into temporal logic assertions and subsequently synthesized with tools such as those described in [2, 20], the size of the resultant logic is often prohibitive for our context. Indeed, the logic implementation of a checker implementing a golden model for a microarchitectural block is often as large as the block itself, and such vast overhead is not tolerable for large blocks. Recent research has focused on reducing logic overhead by sacrificing checking accuracy [63], but did not consider the benefits of complementing that approach with signal tracing.

The possibility of adopting conventional software testbenches for acceleration and emulation platforms has been considered in prior work as well. The testbench still executes in software and communicates with the platform over a bus: in this setup the communication

often becomes the bottleneck [67, 56]. Transaction-based acceleration (TBA) [85] attempts to overcome this bottleneck by bundling several interactions between the testbench and the platform into larger, yet less frequent, transactions.

On the data logging front, acceleration and emulation platforms permit recording the values of a pre-specified group of signals [95], which can be later verified for consistency by a software checker. Recently, a solution was proposed for adapting an architectural checker for a complex processor design to an acceleration platform [27] using this approach: low overhead embedded logic produces a compressed log of architectural events, which is later checked by an off-platform software checker. However, an architectural checker cannot provide the level of insight on design correctness, which a number of local checkers for microarchitectural blocks can. At the architectural level, the information gathered is limited to events modifying the architectural state of the processor; in contrast, microarchitectural checkers track events occurring in individual microarchitectural blocks, generally entailing many more signals. Hence, adapting several microarchitectural checkers provides a much greater challenge, but it is much more rewarding from design debugging perspective.

## 6.7 Summary

This chapter presented a solution to bring in similar degree of checking capability as available in software-based simulation to hardware-accelerated platforms. This solution was demonstrated for a modern microprocessor design. Our solution leverages a combination of local assertions, data compression hardware and off-platform post-simulation analysis for checking complex functionality. We found that our solution is effective in delivering high quality bug detection capabilities at low recording rates (15-25 bits/cycle) and logic overhead ($<$25%) on typical micro-architectural blocks. Such low recording rate and logic overhead will be able to retain the performance advantage of hardware-accelerated simulation platforms. This chapter presented the final contribution of this dissertation; the next chapter will provide a conclusion to this dissertation along with the possible future directions.

# Chapter 7
# Conclusions

The goal of this dissertation was to advance the field of simulation-based verification. Digital designs face increasing complexity and shorter release schedules, challenging the ability of the current design process to deliver a correctly functioning product in a given timeframe. Simulation-based validation is the primary method deployed in the industry to ensure design correctness. The effectiveness of the current state of simulation-based validation is compromised by a critical gap; the most predominant mode of simulation, namely software-based simulation has excellent checking and debugging capabilities but falls short in performance. In contrast, hardware-accelerated platforms offer excellent simulation performance but are crippled by very limited checking and debugging capabilities.

This dissertation bridges this gap from both ends by presenting novel solutions to deliver low-cost high-performance software-based simulation, as well as solutions to provide checking and debugging capabilities on hardware-accelerated simulation platforms. These solutions will enable verification practitioners to harness the potential of simulation to its full extent; design checking and debugging will be achieved at much higher simulation performance than attainable currently. Ultimately this will enable higher validation coverage for shorter product cycles, thereby delivering high quality integrated circuit designs while conforming to tight release schedules. If we can continue the current trends in complexity growth without compromising design correctness, attained through effective high-performance validation, then we can unlock further growth in the semiconductor industry.

## 7.1 Summary of the contributions

This dissertation first presented solutions to improve the performance of software-based simulation at multiple abstraction levels, by leveraging the massive parallelism of GP-GPUs in Chapter 3. **GCS** brings in an order of magnitude improvement in the performance

of gate-level simulation, while **SAGA** improves the performance of simulation at the behavioral level, namely SystemC RTL. These solutions bring the performance of software-based simulation to levels previously offered exclusively by dedicated hardware-accelerated simulation platforms.

Several solutions are also presented to bridge the gap from the hardware-accelerated platform's end, by bringing in checking and debugging capability to these platforms. Signal observability is crucial for debugging, but observability is scarce in platforms beyond software simulation. An approach to solve this problem involves reconstruction of non-observed signals from a small number of observed ones. An **automatic signal selection** solution, with the objective of maximizing the restoration of non-recorded state values from recorded state values was presented in Chapter 4.

Performing design checking on hardware-accelerated platforms while maintaining the performance advantage is a challenging proposition, since, additional logic dedicated for checking, as well as tracing signal values for post-simulation checking, come at a performance overhead. Two major directions to bring in checking capability to hardware-accelerated simulation platforms were explored: i) **Checker approximation** is a solution to bring in checking functionality to these platforms with light-weight embedded logic dedicated for checking; logic overhead for such checking constructs is reduced by trading off checking accuracy ii) **On-platform compression** is a solution used to reduce the volume of data that needs to be traced to perform checking. Both of these solutions were presented in Chapter 5.

Finally, this dissertation culminates in a unifying solution which brings together these two approaches on performing checking on hardware-accelerated platforms for modern microprocessor designs. **Hybrid checking** combines the ideas of using lightweight embedded logic to perform checks during simulation as well as performing post-simulation checks on a compressed event log in a synergistic fashion. This solution was presented in Chapter 6.

### 7.1.1 Infusing performance into software-based simulation

Software-based simulation already possesses excellent design checking and debugging infrastructure, due to decades of research and development. This infrastructure can be easily adapted to a new software-simulator offering better performance, without any major change in methodology. Hence performance improvement of software is extremely beneficial to verification engineers. This dissertation presented solutions to improve the performance of software-based simulation at two design abstraction levels by leveraging a massively parallel execution substrate, namely GP-GPUs.

**GCS** is a gate-level simulator architecture that leverage the high degree of parallelism of GP-GPUs. By extracting the parallelism available in the simulation of gate-level netlists, we were able to achieve an order-of-magnitude speedup over traditional sequential simulators, on average. This simulator was developed in two flavors: oblivious and event-driven. The oblivious version of the simulator maps the parallelism available in netlists to the execution parallelism of GPUs by employing a novel clustering and balancing algorithm. While the event-driven version carves out macro-gates from the structural netlist of a design and schedules them for simulation on the multiprocessors of the GPU, only if they are activated by switching events at their inputs.

**SAGA** is a solution that demonstrates the viability of GP-GPUs as an accelerator for software-based simulation at a much higher design description level, namely SystemC RTL. This problem is more challenging as the computation pattern is even more irregular compared to gate-level simulation. To tackle this challenge, we proposed novel static dataflow partitioning algorithms to extract the parallelism present in the problem to map it to the parallelism available in GP-GPUs. This scheme allows us to forgo frequent synchronizations and deliver better simulation performance. We achieved up to an order-of-magnitude speedup over conventional SystemC simulators.

### 7.1.2 Bringing in debug capability

One of the major roadblocks in debugging beyond software-based simulation is the scarcity of observability. In acceleration and emulation platforms observability often comes with a prohibitive performance cost, while in post-silicon, this problem is even more acute, as dedicated hardware structures are necessary to record signal values. As a result, we can afford to record only a small number of signals; hence, researchers have sought solutions that reconstruct non-observed signals from observed ones. This has resulted in an effort toward developing automatic signal selection algorithms that attempt to choose those flip-flops whose values, if known, lead to the reconstruction of a maximal number of other state values. The performance of these algorithms is measured by the metric of state restoration ratio: the ratio of restored state values vs. the number of recorded state values.

**A novel automatic signal selection algorithm** is presented in this dissertation. This algorithm is general in the sense that it can be applied to any sequential circuit without any specific design knowledge. The selection algorithm is guided by an accurate simulation-based restoration capacity metric and achieves better state restoration ratio than previous solutions. It also achieves better trends of restoration per additional traced signal while restoring higher average number of states. Overall, this solution provides a higher de-

gree of observability into the design for debugging purposes via restoration, than previous solutions in that space.

### 7.1.3 Bringing in checking capability

The rest of the contributions are towards bringing in checking capability in hardware-accelerated platforms while maintaining simulation performance. These platforms are designed to carry out high-performance simulation of synthesized digital logic and do not provide capabilities for checking constructs. Hence, checker-centric validation, although very successful for software-based simulation, currently does not extend to the realm of acceleration or emulation. This dissertation presented two different approaches and finally a unifying methodology to adapt checker-centric validation to these platforms. These approaches were demonstrated on microprocessor designs and the methodology was also developed for microprocessor designs. However, these solutions can be applicable to other classes of designs as well.

**Checker approximation** trades off logic complexity with bug detection accuracy by leveraging novel techniques to approximate software checkers into small synthesizable hardware blocks, which can be simulated along with the design on a hardware-accelerated simulation platform. A generalized checker taxonomy was presented, which proposes a range of approximation techniques based on a checker's characteristic and provides metrics for evaluating its bug detection capabilities. The case study on a microprocessor-like design demonstrated that checker approximation is a viable solution to reduce hardware overhead of complex checkers while still enabling detection of a large fraction of bug manifestations.

**On-platform compression** was demonstrated with a novel microprocessor design checking scheme on acceleration platforms that provides architectural checking against a golden model. On-platform logic was used to trace and compress necessary data for checking, reducing the number of signals to be traced, which is key to retaining the performance advantage of hardware-accelerated platforms. The solution provides the same bug detection quality as its software-based counterpart. It enables architectural validation of the design on acceleration platforms with negligible accuracy loss. Thus it makes microprocessor design validation possible at an order-of-magnitude better simulation performance than software-based simulation.

**Hybrid checking** strives to bring to hardware-accelerated platforms a degree of checking capabilities similar to those available in software-based simulation. This solution leverages a combination of local assertions, data compression hardware and off-platform post-simulation analysis for checking complex functionality. We found that the solution

is effective in delivering high quality bug detection capabilities at low recording rates and permissible logic overhead over a broad range of micro-architectural blocks. Such low recording rate and logic overhead will be able to retain the performance advantage of hardware-accelerated simulation platforms, and yet provide checking capabilities comparable to that of software-based simulation.

## 7.2 Directions of future research

The dissertation opens the door to several future research directions. Design simulation will continue to be the primary mode of validation in the foreseeable future, and simulation performance will continue to be valued among verification engineers. While processors continue to show the trend of integrating many general-purpose processor cores, as well as different types of accelerators (such as GPUs) on the same chip. As the GCS and SAGA solutions have already demonstrated, the simulation algorithm can be mapped to fit the execution parallelism of non-conventional processors. Further research is needed to target the heterogeneous concurrent architecture of the future – finding the right balance on where to map each of the various components of the simulation process, based on their inherent concurrency.

The use of acceleration and emulation platforms is projected to increase in future and this will necessitate the deployment of checking and debugging solutions similar to the ones described in this dissertation. This dissertation demonstrated that a combination of low logic footprint embedded checkers and off-line checkers that operate on compact simulation trace are capable to bring in checking and debugging capability comparable to software-based simulation, on hardware-accelerated platforms. However, the solutions were achieved through manual partitioning and checker re-design and no standard EDA tool flow for this purpose currently exists. Hence, future research will need to devise a standard tool flow for developing checkers for design simulation on hardware-accelerated platforms. Also the checking solutions discussed in this dissertation focused mostly on microprocessor designs, but as System-on-Chip (SoC) designs are becoming predominant, future research will need to bring these solutions to the system level.

# Bibliography


[1] Intel(R) Pentium(R) Processor Invalid Instruction Erratum Overview, July 2004. www.intel.com/support/processors/pentium/sb/cs-013151.htm.

[2] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proc. CAV*, pages 538–542, 2000.

[3] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, pages 7–12, 2006.

[4] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-Pro: Innovations in test program generation for functional processor verification. *IEEE Design and Test of Computers*, 21(2):84–93, 2004.

[5] Advanced Micro Devices, Inc. *Revision Guide for AMD Athlon(TM) 64 and AMD Opteron(TM) Processors*, Aug. 2005.

[6] Altera Verification Tool. *SignalTap II Embedded Logic Analyzer*, 2006. http://www.altera.com/products/software/products/quartus2/verification/signaltap2/sig-index.html.

[7] ARM limited. *Embedded Trace Macrocells*, 2007. http://www.arm.com/products/solutions/ETM.html.

[8] T. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proc. MICRO*, pages 196–207, 1999.

[9] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst-case design. In *Proc. ASPDAC*, 2005.

[10] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 1997.

[11] W. Baker, A. Mahmood, and B. Carlson. Parallel event-driven logic simulation algorithms: Tutorial and comparative evaluation. *IEEE Journal on Circuits, Devices and Systems*, 1996.

[12] Z. Barzilai, J. Carter, B. Rosen, and J. Rutledge. HSS–a high-speed simulator. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 1987.

[13] K. Basu and P. Mishra. Efficient trace signal selection for post silicon validation and debug. In *Proc. VLSI design*, pages 352–357, 2011.

[14] H. Bauer and C. Sporrer. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. *Proc. ANSS*, 1993.

[15] B. Bentley. Validating the intel pentium 4 microprocessor. In *Proc. DAC*, pages 244–248, 2001.

[16] O. Berry and G. Lomow. Speeding up distributed simulation using the time warp mechanism. In *Proc. of workshop on Making distributed systems work*, 1986.

[17] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of The ACM*, 13:422–426, 1970.

[18] M. Boulé, J.-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proc. ICCD*, pages 294 –299, 2006.

[19] M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proc. ICCD*, 2005.

[20] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of psl properties. *ACM Trans. Des. Autom. Electron. Syst.*, 13:4:1–4:21, February 2008.

[21] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: a compiled simulator for MOS circuits. In *Proc. DAC*, 1987.

[22] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35:677–691, 1986.

[23] Cadence. *Palladium*. `http://www.cadence.com/products/sd/palladium_series`.

[24] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor silicon debug based on failure propagation tracing. In *IEEE Transactions on Computers*, pages 10 pp. –293, 2005.

[25] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 1981.

[26] Y.-S. Chang, S. Lee, I.-C. Park, and C.-M. Kyung. Verification of a microprocessor using real world applications. In *Proc. DAC*, pages 181–184, 1999.

[27] D. Chatterjee, A. Koyfman, R. Morad, A. Ziv, and V. Bertacco. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proc. DAC*, 2012.

[28] P. Combes, E. Caron, F. Desprez, B.Chopard, and J. Zory. Relaxing synchronization in a parallel SystemC kernel. In *Proc. Of ISPA*, 2008.

[29] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000.

[30] J. Darringer, E. Davidson, D. Hathaway, B. Koenemann, M. Lavin, J. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan. EDA in IBM: past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1476–1497, 2000.

[31] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of system verilog assertions. In *Proc. DATE*, pages 70–75, 2006.

[32] J. Davis, C. Thacker, and C. C. BEE3: Revitalizing computer architecture research. Technical report, April 2009.

[33] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. BackSpace: formal analysis for post-silicon debug. In *Proc. FMCAD*, pages 1–10, November 2008.

[34] M. Denneau. The Yorktown simulation engine. *Proc. DAC*, 1982.

[35] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proc. ICCD*, pages 522–525, 1992.

[36] W. Ecker, V. Esen, L. Schonberg, T. Steininger, M. Velten, and M. Hull. Impact of description language, abstraction layer, and value representation on simulation performance. In *Proc. of DATE*, 2007.

[37] EDALab. *HIFSuite*, 2011. `http://www.hifsuite.com/`.

[38] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. MICRO*, 2003.

[39] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, and D. Ravi. Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In *Proc. of PADS*, 2009.

[40] H. Foster. DAC 2012 Post-silicon Workshop. `https://www.research.ibm.com/haifa/images/dac/Harry_2012-DAC-Post-Silicon-Workshop.pdf`.

[41] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. In *Proc. DATE*, pages 434–441, March 1999.

[42] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 1990.

[43] G. Ganapathy, R. Narayan, C. Jorden, M. Wang, and J. Nishimura. Hardware emulation for functional verification of K5. In *Proc. DAC*, pages 315 –318, 1996.

[44] K. Gulati and S. Khatri. Towards acceleration of fault simulation using graphics processing units. *Proc. DAC*, 2008.

[45] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang. Visibility enhancement for silicon debug. In *Proc. DAC*, pages 13–18, 2006.

[46] Intel. Nehalem-EX, 2009. `http://download.intel.com/pressroom/pdf/nehalem-ex.pdf`.

[47] Intel. Intel 82574L Ethernet controller bug, 2013. `http://www.theregister.co.uk/2013/02/06/packet_of_death_intel_ethernet/`.

[48] Intel Corporation. *Intel(R) StrongARM(R) SA-1100 Microprocessor Specification Update*, Feb. 2000.

[49] Intel Corporation. *Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, September 2007.

[50] Intel Corporation. *Intel Core i7-900 Desktop Processor Series Specification Update*, July 2010.

[51] International Business Machines Corporation. *IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice*, July 2005.

[52] N. Ishiura, H. Yasuura, and S. Yajima. High-speed logic simulation on vector processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(3):305–321, May 1987.

[53] D. Josephson. The manic depression of microprocessor debug. In *IEEE Transactions on Computers*, pages 657 – 663, 2002.

[54] Khronos Group. `"http://www.khronos.org/opencl/"`.

[55] H. Kim and S. Chung. Parallel logic simulation using time warp on shared-memory multiprocessors. *Proc. IPPS*, 1994.

[56] Y.-I. Kim and C.-M. Kyung. Tpartition: Testbench partitioning for hardware - accelerated functional verification. *IEEE Design & Test*, 21:484–493, 2004.

[57] Y.-I. Kim, W. Yang, Y.-S. Kwon, and C.-M. Kyung. Communication-efficient hardware acceleration for fast functional simulation. *Proc. DAC*, 2004.

[58] H. F. Ko and N. Nicolici. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *Proc. DATE*, pages 1298–1303, 2008.

[59] H. F. Ko and N. Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(2):285–297, 2009.

[60] D. Lewis. A hierarchical compiled code event-driven logic simulator. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 1991.

[61] X. Liu and Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Proc. DATE*, pages 1338–1343, 2009.

[62] J. M. Ludden et al. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems. *IBM Journal of Research and Development*, 46(1):53–76, 2002.

[63] B. Mammo, D. Chatterjee, D. Pidan, A. Nahir, A. Ziv, R. Morad, and V. Bertacco. Approximating checkers for simulation acceleration. In *Proc. DATE*, 2012.

[64] N. Manjikian and W. Loucks. High performance parallel logic simulations on a network of workstations. *Proc. of workshop on Parallel and distributed simulation*, 1993.

[65] J. Markoff. Burned once, Intel prepares new chip fortified by constant tests. *New York Times*, Nov. 2008.

[66] Y. Matsumoto and K. Taki. Parallel logic simulation on a distributed memory machine. *Proc. EDAC*, 1992.

[67] I. Mavroidis and I. Papaefstathiou. Efficient testbench code synthesis for a hardware emulator system. In *Proc. DATE*, pages 888–893, 2007.

[68] Microsoft. `http://www.xbox.com/kinect`.

[69] MiniSat. `www.minisat.se`.

[70] J. Misra. Distributed discrete-event simulation. *ACM Computer Survey*, 1986.

[71] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. *Proc. of ASP-DAC*, 2010.

[72] N. Nataraj, T. Lundquist, and K. Shah. Fault localization using time resolved photon emission and STIL waveforms. In *IEEE Transactions on Computers*, pages 254 – 263, 2003.

[73] NVIDIA. *CUDA Compute Unified Device Architecture*, 2007.

[74] Open SystemC Initiative. *SystemC Language Reference*, 2011. http://www.systemc.org/downloads/standards.

[75] Opencores. "`http://www.opencores.org/`".

[76] S.-B. Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1545 –1558, oct. 2009.

[77] P. Patra. On the cusp of a validation wall. *IEEE Design & Test*, 24(2):193–196, 2007.

[78] A. Perinkulam and S. Kundu. Logic simulation using graphics processors. In *Proc. ITSW*, 2007.

[79] V. Popescu and B. McNamara. Innovative verification strategy reduces design cycle time for high-end sparc processor. In *Proc. DAC*, pages 311–314, 1996.

[80] S. Prabhakar and M. Hsiao. Using non-trivial logic implications for trace buffer-based silicon debug. In *Proc. ATS*, pages 131–136, 2009.

[81] R. Raghavan, J. Hayes, and W. Martin. Logic simulation on vector processors. In *Proc. ICCAD*, pages 268–271, Nov 1988.

[82] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Proc. DAC*, pages 445–450, 1998.

[83] N. Saviou, S. Shukla, and R. Gupta. *Design for Synthesis, Transform for Simulation: Automatic Transformation of Threading Structures in High Level System Models*. University of California at Irvine, 2008. Technical Report TR-01-58.

[84] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, Aug. 2008.

[85] M. Shabtay, D. Leonard, B. Maya, and S. Michael. Building transaction-based acceleration regression environment using plan-driven verification approach. In *Design and Verification Conference and Exhibition*, 2007.

[86] H. Shojaei and A. Davoodi. Trace signal selection to enhance timing and logic visibility in post-silicon validation. In *Proc. ICCAD*, pages 168–172, 2010.

[87] G. Spirakis. Opportunities and challenges in building silicon products in 65nm and beyond. In *Proc. DATE*, 2004.

[88] Sun microsystems OpenSPARC. "`http://opensparc.net/`".

[89] Synopsys. *Identify Pro*. `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/Identify.aspx`.

[90] Synopsys. Synopsys Magellan. "`http://www.synopsys.com`".

[91] B. Turumella and M. Sharma. Assertion-based verification of a 32 thread SPARC CMT microprocessor. In *Proc. DAC*, 2008.

[92] B. Vermeulen, T. Waayers, and S. Bakker. IEEE 1149.1-compliant access architecture for multiple core debug on digital system chips. In *IEEE Transactions on Computers*, pages 55 – 63, 2002.

[93] D. W. Victor et al. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM Journal of Research and Development*, 49(4):541–554, 2005.

[94] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification*. Morgan Kaufmann Publishers Inc., 2005.

[95] Xilinx Verification Tool. *ChipScope Pro*, 2006. `http://www.xilinx.com/ise/optional_prod/cspro.html`.

[96] J.-S. Yang and N. A. Touba. Automated selection of signals to observe for efficient silicon debug. In *Proc. VTS*, pages 79–84, 2009.

[97] J.-S. Yim, Y.-H. Hwang, C.-J. Park, H. Choi, W.-S. Yang, H.-S. Oh, I.-C. Park, and C.-M. Kyung. A C-based RTL design verification methodology for complex microprocessor. In *Proc. DAC*, 1997.

[98] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun. A parallel SystemC environment: ArchSC. In *Proc. of ICPADS*, 2009.